AI-Driven Solo Automation SaaS Project Plan

AI-Driven Automation Platform – Project Plan

Introduction

Many small and mid-sized businesses (SMBs) struggle with **manual, error-prone administrative tasks** across various domains, from **fleet compliance document management** to **contract renewals**. An analysis of automation pain points (see *AutomationPainPointSolutions.pdf*) identified several niche workflows ripe for automation – for example: tracking fleet permits, abstracting commercial lease dates, automating lien waiver exchanges in construction, preparing workers' comp audits, coordinating building permits/inspections, tracking vendor insurance certificates, auditing utility bills, logging environmental compliance, and monitoring contract renewals. These processes are often fragmented and handled with spreadsheets or email, leading to missed deadlines and inefficiencies. This project aims to build a **unified AI-driven automation platform** to tackle these pain points. The platform will have a **canonical core schema** (common data model and workflow engine) that can be **configured via niche-specific YAML templates** to handle each of the above scenarios. By leveraging AI (OCR and Large Language Models) for document processing and using modern cloud tools, even a solo founder can develop and launch this as a Software-as-a-Service (SaaS).

Opportunity & Vision: There is *massive demand* for niche automation tools that solve specific problems, and today's technology lets a single developer meet this demand at low cost. Powerful pre-trained AI models (like GPT-4) can be tapped via API without having to build models from scratch, and no-code/low-code platforms can accelerate development. The goal is to deliver an MVP (Minimum Viable Product) within 90 days that **solves one selected pain point extremely well** – following the successful pattern of solo SaaS products that focus on a *"single pain point, simple interface"*. This focused approach will allow quick validation and user feedback. Once the core platform is proven on one niche, it can be rapidly extended to other niches by adding new YAML templates (rather than rewriting software), thereby scaling the business.

Why a Solo AI-Powered Approach: Thanks to AI tools acting as "team members," a single person can handle roles that used to require a full team. For instance, ChatGPT can serve as a coder, tester, copywriter, and even marketing strategist all in one. This project will extensively use such AI assistants for development, content creation, and even customer support, effectively multiplying the founder's productivity. Modern examples of one-person AI startups underscore the viability of this approach – many founders have reached significant revenue by leveraging AI and APIs instead of hiring large teams. In short, the time

is ideal to build a one-human, AI-driven business, keeping costs low and execution speed high.

Roles and Responsibilities

Because this is a **solo venture**, the founder will wear many hats. Below are the key "roles" and how responsibilities are allocated. (AI tools will augment each role as described):

- **Product Manager (Founder):** Defines project scope and priorities, selects the initial niche to target, and translates pain points into features. Responsible for breaking down the 90-day plan into actionable tasks and ensuring the project stays on track. *AI Assistance:* Use ChatGPT for brainstorming requirements and solution ideas, and for generating user stories or acceptance criteria.
- **Lead Developer (Founder):** Designs the system architecture and writes code for both the frontend and backend. Handles integration of AI services (OCR/LLM) and ensures the core schema and YAML-driven config system are implemented. Also sets up cloud infrastructure and deployment. *AI Assistance:* Utilize GitHub Copilot and ChatGPT for coding help (boilerplate, debugging) to accelerate development. Use AI for code reviews and optimizing algorithms.
- **AI/ML Specialist (Founder):** Develops the NLP/OCR pipeline for parsing documents and extracting data. Fine-tunes prompts or models as needed for each niche (e.g., extracting dates from leases or insurance forms). *AI Assistance:* Leverage existing APIs (OpenAI for language tasks, OCR libraries or Google Vision API for text extraction) rather than building models from scratch. Use AI tools to experiment with prompt engineering and to quickly prototype ML workflows.
- **UI/UX Designer (Founder):** Designs a simple, intuitive web interface for the application. Chooses a clean layout to input data, upload documents, and display alerts (one that can be easily skinned per niche if needed). *AI Assistance:* Use tools like Midjourney for brainstorming logo and graphic ideas, and Figma with AI plugins or pre-made templates for quick UI designs. Leverage AI image generation to create icons or illustrations if needed.
- **QA Engineer (Founder):** Tests the application in each development sprint. Writes test cases to cover core functionality and uses automated testing where possible. *AI Assistance:* Use AI to generate test scenarios (e.g., "what are edge cases for document upload?"), and potentially use AI-driven testing tools to simulate user interactions. AI can also assist in security testing by suggesting potential vulnerabilities.

- **Marketing & Customer Support (Founder):** Even before launch, responsible for marketing the product (creating a landing page, blogging, social media posts) and handling user onboarding and feedback. *AI Assistance:* ChatGPT will be used to draft marketing copy, blog posts, and tutorial content. AI can also help create a knowledge base or FAQ by summarizing how the system works. For support, an AI chatbot could be integrated to answer common user questions in real-time, giving the appearance of a 24/7 support team even with one person behind the scenes.

Note: While the founder is primary for all roles, this plan doesn't rule out selectively outsourcing small tasks if needed. For example, if a polished UI is a priority, the founder might purchase a UI template or hire a freelance designer for a day. However, given budget constraints, the emphasis is on using AI and existing resources over hiring. The budget will mainly go towards cloud services and API calls, not salaries. With AI's help, the founder can realistically manage coding, design, and marketing simultaneously – as evidenced by real-world solo startups where AI acted as the equivalent of coders, designers, and marketers.

Technical Stack and Architecture

Choosing the right tech stack is critical for speed and scalability. We will favor technologies that are **high-productivity, well-supported, and integrate easily with AI components**. Below is the proposed stack for this project:

- **Backend: Python** – Preferred for its rich ecosystem in automation and AI. The backend will be built with a Python web framework (likely **FastAPI** for a lightweight, fast API, or **Django** for a more batteries-included approach if an admin interface is needed). Python offers libraries for OCR (e.g., Pytesseract) and NLP, and it's easy to call external AI APIs (OpenAI, etc.) with Python SDKs. The backend will expose a RESTful (or GraphQL) API that the frontend can consume. *Rationale:* Python's versatility and extensive library support allow quick integration of machine learning tasks and rapid development. Many solo AI SaaS projects use Python for these reasons.

- **Frontend: React.js** – A dynamic single-page application (SPA) front-end using React (possibly with **Next.js** for ease of setup and SSR). React is chosen for its component reusability and the vast array of UI libraries (Material-UI, Ant Design, etc.) that can accelerate building forms, tables, and dashboards. The UI will be responsive so that it works on mobile browsers as well (important if users want to check statuses on the go). The frontend will communicate with the backend via API calls (HTTPS/JSON). *Alternative:* If time is very tight or front-end complexity is a bottleneck, consider using a **no-code front-end tool** or an admin template for the

MVP. For example, **Retool** or **Bubble** could stand up a basic interface quickly, though with less flexibility than custom React.

- **Database: PostgreSQL** (SQL relational database) – Chosen for reliability and its ability to handle complex relational queries, which we'll need for the core schema (linking entities, documents, tasks, etc.). A hosted service like Amazon RDS, Heroku Postgres, or **Supabase** (which is Postgres under the hood with a nice API) will be used to avoid managing DB servers. PostgreSQL's JSONB support could also store semi-structured data or the YAML-defined custom fields for each niche in a flexible way if needed. Alternatively, for quick start, even **Firebase Firestore** (NoSQL) could be used given its easy integration with front-end and offline support – but the structured nature of our data (and potential need for SQL joins across entities) makes Postgres a better fit.
- **Authentication & User Management:** Use a pre-built solution to save time – e.g., **Firebase Auth** or **Auth0** – to handle user sign-up, login, password resets, and possibly multi-tenant account separation. This prevents spending weeks on security boilerplate. Both have React SDKs for easy integration. If using Django, its built-in auth can work too, but third-party auth-as-a-service simplifies OAuth/social logins if needed.
- **Storage:** For storing uploaded documents (like insurance PDFs, lease scans, etc.), use cloud storage (AWS S3 or Google Cloud Storage). This will hold the files which the OCR/LLM pipeline will process. We'll store file metadata and extracted data in the database, but the file itself in storage for scalability. Using a service ensures reliability and CDN delivery for any files that need to be accessed by users.
- **AI and Automation Services:** We will integrate several external APIs for AI capabilities:
    - **OCR**: Use an OCR engine to extract text from uploaded scans. For MVP, **Tesseract** (open-source OCR) can run on the server for cost efficiency. If accuracy or handwritten text becomes an issue, consider **Google Vision API or AWS Textract** for more robust OCR.
    - **LLM/NLP**: Use **OpenAI's GPT-4** (or GPT-3.5) via API to parse and interpret text. For example, after OCR extracts raw text from a lease or insurance document, an LLM can identify key fields (expiration dates, amounts, names) using prompt-based extraction. (We will craft prompts that instruct the LLM to output JSON with the required fields based on the document text.) This approach avoids needing to train a custom model initially. For simpler

structured documents (like standard forms), we might use rule-based parsing or regex as a fallback to minimize API calls.

    - **Email/SMS Notifications**: Integrate with an email service (SendGrid, Mailgun) to send out alerts and reminders. Possibly also SMS via Twilio if needed for urgent reminders (e.g., "Permit expires tomorrow!"). These services have Python libraries for easy use.
    - **Automation/Webhooks**: Use a tool like **Zapier or Make.com** for any external integrations if needed (for example, automatically adding calendar events for deadlines, or integrating with third-party systems). This avoids custom-coding every integration. The platform can emit webhooks for certain events (e.g., "document X is due in 1 day") and Zapier can catch that to trigger other actions in the user's environment (if needed).

- **Hosting & Deployment:** Aim for a cloud platform with minimal DevOps. Possible choices: **Heroku** for its simplicity (good for a Python/Node stack and has add-ons for Postgres, etc.), or **Vercel** + **Serverless Functions** (Vercel for front-end hosting and serverless Node functions; though our backend is Python, we could containerize it or use **AWS Lambda** via **Zappa/Serverless Framework**). Another option is **Fly.io** or **DigitalOcean App Platform** which can run a Dockerized Python app easily. The goal is to avoid managing infrastructure – use platforms that auto-scale as needed. Continuous integration can be set up via GitHub Actions to auto-deploy on commit.
- **Architecture Considerations:** We'll follow a modular monolith approach initially – one codebase for the backend that includes modules for each niche (or a parameterized module that reads the YAML). This keeps things simple. The core will implement generic services (user management, document storage, notification scheduler). The niche-specific logic will largely live in configuration, but any custom code (if needed for certain niches) can be added as plugin modules. We will also ensure the front-end is decoupled from back-end via a clean API – this allows, for instance, building a mobile app later that uses the same API.

Development Tools: The team of one will use tools that maximize productivity. GitHub (with Copilot as noted) for version control and AI-assisted coding, Trello or Notion for task tracking (to organize the 90-day plan into sprints), and Postman for API testing. For data modeling and schema design, a visual tool like Draw.io or an ERD generator will be used to map out the canonical schema (see next section). These choices ensure rapid development and easy collaboration with AI assistants or any future contributors.

Platform/Automation Scope: Web vs. Mobile vs. Hybrid

Primary target platform: Browser-based SaaS (web application). Given the nature of the workflows (business documents, forms, dashboards), a web app is the most practical and cost-effective choice initially. Users can access it from any computer, and a responsive design will allow tablet/phone usage as well. Web SaaS is easier to iterate on quickly – updates can be deployed instantly without requiring users to install updates. It also aligns with how most businesses use admin tools (at a desk).

Mobile considerations: While a native mobile app is not in the 90-day MVP, we will design with future mobile support in mind. Some use-cases (like a contractor snapping a photo of a permit at a job site, or a fleet manager checking a dashboard on the road) could benefit from a mobile app. The options for mobile are:

- *Later Native App:* Develop separate iOS/Android apps once core functionality is stable, mainly for push notifications and offline use. This is more effort and likely beyond a solo developer's initial bandwidth.
- *Hybrid App:* Use **React Native** or **Flutter** to build a cross-platform app reusing a lot of the logic/UI concepts from the web. This could be considered in a second phase if there's demand for a dedicated app experience.
- *Progressive Web App (PWA):* An interim solution where the web app itself can be "installed" on mobile home screens and can send push notifications. We can relatively easily make the React app a PWA, giving some app-like capabilities without separate development.

For the MVP, **best practice is to start with a great web experience**. Many successful SaaS products launched on web and only added mobile apps after gaining traction. This keeps the development scope manageable. The web app will be designed to be mobile-responsive, so users *can* use it on a phone browser if needed. We will gather feedback on how often users want a native mobile experience. If mobile usage is significant, we can prioritize building a React Native app in a subsequent 90-day plan.

Integration with other platforms: The browser-based approach also makes it easier to integrate with things like browser automation (if needed for scraping external sites for info) and to plug into other web services. If one of the automation niches eventually calls for a browser extension (for example, to scrape data while user is on a government permit site), that could be an add-on. But for now, a standalone web portal is the plan.

In summary, **launch on web first** (fast global reach, no app store friction, easier updates). Keep the door open for mobile by writing clean API endpoints and using responsive design.

As a solo founder, focusing on one platform initially is crucial – trying to build web, iOS, and Android all at once would spread development too thin and delay launch (*what not to do!*). The chosen tech stack (React + API backend) naturally enables later expansion to mobile or desktop (e.g., Electron or a Mac menu-bar app like some solo products did) if those become high-value additions.

AI Integration Strategy (NLP, OCR, LLM Components)

Incorporating AI is a core differentiator for this platform, enabling automation that goes beyond simple rule-based workflows. From day one, we will **include NLP/OCR/LLM-powered components** to handle unstructured data and complex decisions. Here's the plan for AI integration in the MVP:

- **Document Ingestion (OCR):** Many pain points involve documents (leases, permits, certificates) that may be PDFs or scans. The system will automate data entry from these by reading the documents. We will implement an OCR service so that when a user uploads a document, the text content is extracted automatically. Initially, we can call a cloud OCR API for high accuracy. If budget is a concern, we use Tesseract locally and improve as needed. This saves users time and reduces errors by eliminating manual data entry from paperwork.
- **Data Extraction & Understanding (LLM/NLP):** After OCR, the raw text needs to be understood. We will use **Large Language Models** to parse and extract structured information. This is essentially using GPT-4 (via OpenAI API) as a flexible parser that can handle varied document formats:
    - *Example:* For a **Commercial Lease**, we'd prompt GPT-4 with: "Extract the following fields from this lease text: tenant name, lease start date, lease end date, renewal notice deadline, annual rent, escalation rate… Output as JSON." The model's output is then parsed into our database fields.
    - *Example:* For a **Certificate of Insurance (COI)**, prompt for fields like policy expiry date, coverage amounts, insurer name, etc. The LLM can interpret the form even if it's a scan (after OCR).
    - Using an LLM in this way means we don't have to write custom parsing code for each document template – the AI can generalize. It accelerates development significantly, though we'll need to validate the results. We'll employ confidence checks (e.g., if a critical field is missing or the AI is not certain, flag for human review).

    - We might start with GPT-4 for development due to its reliability, but also test open-source models (like LLaMA 2 or others) if we need to self-host for cost reasons in the future.

- **Natural Language Queries & Summaries:** As a value-add, the platform can include AI-driven user queries. For example, a user could ask the system in plain English, *"Which permits are due next month?"* or *"Summarize this lease in 5 bullet points."* An LLM can be used to interpret the question and either fetch the answer from the data (for the first question) or generate a summary (for the second). While not core to the MVP, this feature aligns with the vision of an AI assistant built into the product. We will design the system such that each niche module could have an "Ask AI" button for common questions (the heavy lifting done by GPT under the hood, with safeguards).
- **Intelligent Notifications:** Rather than static date-based reminders, we can use AI to enhance notifications. For example, if a contract's renewal is coming up, the system might draft an email to the client for the user. Using an LLM, the system could generate a first draft of a renewal notice or a gentle reminder, which the user can review and send. This saves the user time composing messages and leverages AI's copywriting ability. Similarly, if a compliance issue is detected (say an expired document), the system could generate a brief report explaining the issue and potential steps to resolve it, not just a raw alert.
- **Learning from Data (Future):** As the system accumulates data (across various niches and documents), we could consider training niche-specific models or fine-tuning an LLM on a knowledge base of regulations (for example, upload OSHA guidelines and let the AI answer compliance questions). This is beyond the 90-day scope but is an exciting direction to truly make the platform an intelligent assistant for each domain. Even without custom training, we can use embeddings and similarity search so that the AI can reference relevant info (like pulling a clause from a contract when asked).
- **AI for Internal Development:** It's worth noting that AI will also be used internally during development (as mentioned in roles) – e.g., generating test data by having ChatGPT create sample leases or dummy vendor certificates, which helps in testing our parsing logic. This makes QA faster and more robust.

Ensuring Accuracy and Reliability: One challenge with AI (especially LLMs) is occasional errors or "hallucinations." We mitigate this by:

- Keeping humans in the loop for critical extractions: highlight the extracted fields and show them alongside the original text to let the user verify. For instance, after parsing a lease, show the key dates detected with the snippet of text around each date for context.
- Using validation rules: e.g., if an expiry date extracted is in the past, that's likely wrong – flag it. Or if an amount supposed to be yearly rent is extracted as a very small number, double-check units.
- Starting with well-defined uses of AI (structured extraction, summarization) rather than open-ended generation. This confines the model to tasks where accuracy can be more easily checked.
- Logging all AI outputs and user corrections – this could later be used to fine-tune a model or at least to continuously improve prompt engineering.

By **including NLP/OCR from the start**, the product will deliver a "wow" factor (automating hours of tedious reading/data entry). At the same time, by using existing AI services and not trying to invent new algorithms, we keep development fast and costs low. This combination of off-the-shelf AI and careful engineering will enable a powerful MVP within 90 days.

Canonical Core Schema

At the heart of the platform is a **canonical core schema** – a generic data model that can represent various entities and workflows across different niches. Designing this schema is an early priority, as it ensures that adding new niches is mostly a matter of configuration (not redesigning the database each time). Below are the main components of the schema and how they relate:

- **Entity (Subject):** This represents the primary object that the compliance or task is associated with. It could be a *Vehicle* (for fleet compliance), a *Property/Lease* (for lease management), a *Project* (for construction lien waivers), a *Vendor* (for COI tracking), a *Contract* (for renewal tracking), etc. Rather than separate tables for each type, we can have a single **Entities** table with a **type** field (or a subtype table for each if needed). Common fields: unique ID, name/identifier (e.g., vehicle ID, property address, vendor name), and a link to the owning **Account/User**. For example, an Entity of type "Vendor" might have name = ACME Supplies, and be linked to user = John's Construction Co.
- **Requirement/Task:** This table defines a **compliance requirement or recurring task** associated with an entity. For instance, "Annual vehicle registration renewal" could be a requirement linked to a Vehicle entity; "Monthly utility bill audit" linked to

a Property entity; “Lien waiver submission for April payment” linked to a Project entity. Fields might include: requirement ID, entity ID (foreign key to Entity), **type** of requirement (e.g., ‘Vehicle Registration’, ‘Lease Expiry Alert’, ‘Lien Waiver’ – could also be a foreign key to a Requirement Type template table), status (Pending, Completed, Overdue, etc.), due date, frequency (if recurring), and assigned user (who is responsible, if needed). This is a central table for tracking what needs to be done or monitored.

- **Document:** A table to store metadata of any documents uploaded or generated. It could include: document ID, linked entity or requirement (e.g., an insurance certificate document linked to a Vendor entity), document type (e.g., ‘Insurance Certificate’, ‘Lease Agreement’, ‘Permit PDF’), storage path or URL (to S3, etc.), upload date, and possibly a JSON field for **extracted data** from the document (the parsed fields via OCR/LLM). By storing extracted data in JSON, we keep flexibility for different document types (one type might have “expiryDate” and “policyNumber”, another might have “startDate”, “endDate”, etc.). There may also be a flag for verification (e.g., “verified true/false” if a human has confirmed the extraction is correct).
- **User & Account:** Standard tables for user authentication and grouping. Likely we’ll have **Users** and an **Account/Organization** table, since a single company might have multiple user logins collaborating. For MVP, if targeting small clients, a simple user table might suffice (each account is separate and has one user). But thinking ahead, we include an account ID on entities to segregate data by client. The user table includes name, email, hashed password (if not using external auth), role (maybe ‘admin’ vs ‘read-only’ if needed in future). The Account table might hold company name, subscription plan, etc.
- **Notification/Alert:** A table to log any notifications or alerts that are scheduled or have been sent. Fields: notification ID, related requirement ID (or document ID), type (email, SMS, in-app), date to send, date actually sent, status (pending, sent, acknowledged). This helps manage the scheduling of reminders. For example, when a requirement is created with a due date, the system might create a notification entry for “send reminder 30 days before due” and “send alert on due date if not completed”. A background service or cron job will check this table daily to dispatch notifications at the right time.
- **Audit Log:** To track changes and user actions (optional in MVP but good to have for compliance-related software). This could record events like “user X marked requirement Y as completed at time Z” or “system parsed document D and

extracted field value changes." This is useful for debugging and for users to have an activity history.

- **Template Definitions (Lookup tables):** There will be a set of tables that define the templates loaded from YAML. For instance:
    - **RequirementType** table: defines possible requirement types (e.g., "Fleet Registration Renewal", "COI Expiration Tracking", "Workers Comp Audit") with fields like default frequency, description, etc. The YAML for a niche will populate this with that niche's specific tasks.
    - **DocumentType** table: defines types of documents (e.g., "Insurance Cert", "Lease Agreement", "Lien Waiver Form") along with maybe which fields to extract (this could be a JSON schema or a reference to another structure defined via YAML).
    - **Field Definitions (optional):** If we want to store structured extracted data in a relational way, we could have tables for each field type or at least a metadata table. However, this might over-complicate the initial design. We can rely on JSON in the Document record for extracted key-value pairs, and only promote certain universal fields to top-level columns if needed for queries (e.g., a next renewal date could live in Requirement for easy querying).
- **Relationships:** Key relationships include:
    - Account has many Users.
    - Account has many Entities.
    - Entity has many Requirements (or compliance tasks).
    - Requirement may have a related Document (or multiple documents) – e.g., a "COI Tracking" requirement would link to the latest COI document.
    - Entity or Requirement can have many Documents (like an entity "Property" could have lease document, inspection reports, etc., or requirement "Monthly Report" might produce a document each month).
    - Requirements have many Notifications (one requirement can spawn multiple reminder notifications).
    - All key tables have an Account/Owner link to enforce data isolation between different client accounts in the SaaS.

Schema Example: For illustration, if the chosen initial niche is *Vendor Insurance (COI) Tracking*, the data might look like:

- Entity: Vendor (name: ACME Supplies, account: ConstructionCo Ltd).
- Requirement: "General Liability Insurance" for that Vendor, due date = 2024-12-31, status = Pending.
- Document: ACME's insurance certificate PDF, linked to that requirement (or vendor), DocumentType = "COI", extracted data JSON = { policyNumber: 12345, expiryDate: 2024-12-31, insured: "ACME Supplies" }.
- Notification: scheduled 30 days before 2024-12-31 to email the vendor or the account admin to renew the policy.
  All of the above fits into the generic tables with appropriate type labels.

The **canonical schema ensures that adding a new workflow** (say "Fleet Inspection Reminder") doesn't require a new table – we add a new RequirementType ("Fleet Inspection") and the data (like inspection due date) gets stored as a Requirement with maybe a Document for the inspection report. This uniformity is what makes the platform scalable across niches. It does require careful design to avoid awkward fits, but the niches identified have a lot in common (tracking **things** – permits, contracts, licenses – that have **due dates** and **documents** attached). The schema focuses on these commonalities.

We will create an ERD (Entity Relationship Diagram) early in the project to visualize these tables and relations and include it in project docs for clarity. That will guide both the database setup and the writing of the YAML templates to map into this schema.

Niche YAML Template (Config Example)

To allow the platform to adapt to different niches without hard-coding each one, we will use **YAML configuration files** that define each niche's specifics. Each niche YAML will declare the types of entities, documents, and requirements relevant for that domain, as well as any custom logic or fields. The application will load this config to know what to display and how to behave for a given niche deployment. Below is a description of what the YAML contains, followed by an example for a specific niche.

What the YAML defines:

- **Niche name and description:** For example, "Vendor Insurance Tracking" or "Fleet Compliance Manager". This could be used for display or logging.
- **Entity Types:** The primary object(s) in this niche. e.g., EntityType: Vendor with fields like Vendor Name, Contact Person, etc. (If a niche had multiple entities, they can all

be defined – though many niches have one main entity; some might have two, like a lease niche could involve Property and Tenant as entities.)

- **Requirement Types:** A list of compliance items or tasks to track. For each requirement type, define attributes such as:
    - Name (e.g., "Insurance Certificate Expiration").
    - Applicable Entity Type (e.g., Vendor).
    - Frequency or due date rules (e.g., "annual", or "90 days after lease start for first inspection", etc.). This can be a cron-like schedule or a rule relative to an entity field (for instance, if tracking a permit renewal that's due 1 year after last renewal date).
    - Required Document Type (if any) and the key fields to extract from it. For example, link to a DocumentType "Insurance Cert" and specify that from this document we need expiryDate and policyNumber.
    - Notification rules: how far in advance to remind, who to remind (user vs external party), escalation (e.g., if expired and not updated, alert daily).
- **Document Types:** Definitions of document formats and fields. This could include sample regex or keywords to help identify the doc, but since we're using AI, it might just list the fields we expect. E.g., for "Insurance Cert", fields: policyNumber (string), expiryDate (date), insuredName (string), etc. Possibly include a snippet of prompt for the LLM specifically for this doc type.
- **Workflow/Automation rules:** Any special actions that should happen for this niche beyond generic ones. For instance, for "Lien Waiver Automation," the YAML might specify: when a Pay Application is marked paid, auto-generate a lien waiver form document. Or for "Permit Coordination," it might specify an external API to check permit status. These can be expressed as simple conditional rules or references to code hooks (which we could implement as needed).
- **Glossary or Terminology (if needed for AI):** If a niche has domain-specific terms or acronyms, we might include a section in YAML that defines these for the AI prompts (to improve extraction). For example, define that "COI" means "Certificate of Insurance" so the AI is aware. This is optional and can also be handled by prompt engineering externally.

Example YAML – "Vendor Insurance (COI) Tracking":

Let's illustrate a portion of a YAML config for the vendor insurance use-case:

```
niche: "Vendor Insurance Tracking"
entity_type:
  name: "Vendor"
  fields:
    - name: "Vendor Name"
    - name: "Primary Contact"
    - name: "Contact Email"
requirement_types:
  - name: "Insurance Policy Renewal"
    applies_to: "Vendor"
    document_type_required: "Insurance Certificate"
    key_date_field: "expiry_date"        # which field drives the due date
    recurrence: "annual"                 # this means we expect a renewal yearly
    notifications:
      - days_before: 30
        message: "Insurance policy for {{entity.Vendor Name}} is expiring in 30 days (expires on {{requirement.due_date}}). Please obtain an updated COI."
        notify: "owner"   # notify the account owner
      - days_after: 0
        message: "Insurance policy for {{entity.Vendor Name}} has expired! (expired on {{requirement.due_date}}). No current COI on file."
        notify: "owner"
document_types:
  - name: "Insurance Certificate"
    fields:
      - name: "policy_number"
      - name: "insured_name"
```

```
    - name: "expiry_date"
    - name: "coverage_amount"
  ai_extraction:
    prompt_template: |
      Extract the policy number, insured name, coverage amount, and expiration date from this insurance certificate.
      Format: {"policy_number": "...", "insured_name": "...", "coverage_amount": "...", "expiry_date": "..."}.
```

(The YAML above is an illustration. In practice, the syntax might differ and include more details, but this shows the idea.)

Explanation: In this YAML, we define one entity type "Vendor" with a couple of fields. We then define a requirement type "Insurance Policy Renewal" that applies to Vendor. It expects an "Insurance Certificate" document to be provided. The key_date_field "expiry_date" tells the system that the due date of the requirement is driven by the document's expiry_date field (for instance, once a COI document is uploaded and parsed, the requirement's due date becomes that expiry date – meaning that's when renewal is needed). We set notifications 30 days before and on the day of expiry, addressing the account owner (the user managing vendors). Under document_types, we specify the fields of an Insurance Certificate and even include a prompt template for AI extraction, so the system knows how to ask the LLM for those fields.

When the application loads this config:

- It would create internal representations (in the database or code) for the Vendor entity, the Insurance Policy Renewal task, etc.
- The front-end could automatically generate a form for adding Vendors (with the specified fields), and a form/upload for the Insurance Certificate.
- The back-end uses the ai_extraction.prompt_template when processing an uploaded COI to get the structured data.

Using YAML for configuration has several benefits:

- We can add new niches by writing new YAML files, with minimal code changes.
- It's easier to visualize and discuss the domain logic in a YAML (or JSON) format with domain experts, if needed, compared to hard-coded logic.

- We could even allow power-users to customize or add their own templates in the future using this format (that could be a selling point down the line – e.g., a customer defines a custom compliance process via YAML in a UI).
- During the 90-day project, writing the YAML for the chosen niche will be one of the first tasks (after designing the core schema) to make sure the schema supports what the YAML needs to express.

For the MVP, we'll focus on **one niche's YAML** (e.g., COI Tracking or whichever pain point we target first). We will, however, structure the code to allow switching configs easily. This might mean having a setting or subdomain for the app indicating which config to use, or if we make it multi-niche in one app, allow the user to enable different modules. Initially, it might be simplest to **bake in one YAML** (load it at startup) just to reduce complexity. Then as we expand, we can make it multi-tenant with different YAML per tenant or a selection.

In summary, the YAML template approach provides **niche-specific customization on top of a common platform**. It's a powerful way to scale the product horizontally into multiple domains without reinventing the wheel each time. We will document the YAML schema meticulously and perhaps even write a script to validate the YAML (catch errors like a requirement referring to a document type that isn't defined, etc.). This ensures reliability as we add more configurations.

90-Day Execution Calendar (Block Timeline)

A 90-day (roughly 3-month) timeline is proposed to execute this project from inception to MVP launch. The plan is broken into weekly blocks with specific goals and milestones. Being a solo developer project, the schedule must account for context-switching between roles (development, testing, etc.), so some tasks will run in parallel where possible (with AI assistance making parallelization easier). Below is the execution calendar:

Weeks 1-2: Project Setup and Planning

- **Requirement Refinement:** Finalize which niche to target first (if not already decided). Given the research, likely start with a manageable yet valuable niche (e.g., **Vendor COI Tracking** as MVP module due to its clear scope and demand). Define the MVP feature set for this niche – what must be in Day 90 launch vs. what can wait.
- **Core Schema Design:** Draft the canonical schema ERD on paper/whiteboard. Identify all entities, relationships, and field requirements. Review it against the chosen niche's needs to ensure nothing is missing.

- **Tech Stack Decisions:** Confirm all major tools (framework versions, choose Django vs FastAPI, select Postgres provider, etc.). Set up development environment (IDE, repository, CI pipeline).
- **YAML Template Draft:** Write an initial YAML for the chosen niche (e.g., COI tracking) as per the domain understanding. This will act as a blueprint to test the schema – iterate between schema design and YAML to align them.
- **AI Strategy Plan:** Formulate how exactly documents will be parsed for this niche. For example, gather 2-3 sample insurance certificates to test OCR and parsing. Write sample prompts for GPT-4 and see the output quality (this can be done with the OpenAI playground or a quick Python script). This helps validate our approach early.
  Milestones by end of Week 2: A complete project plan (this document), a finalized schema design (maybe in a README or notion doc), and a sample YAML config. Development environment is ready with a "Hello World" app running (proves that the stack is correctly set up).

Weeks 3-4: Backend Foundation & Database

- **Set Up Database:** Create the database and implement the core schema tables (using migrations if Django/SQLAlchemy if using FastAPI). This includes user accounts, entity, requirement, document, etc., as described earlier.
- **Backend API – Phase 1:** Implement basic CRUD APIs for the core objects: e.g., create/read/update/delete for Entities and Documents. Also implement listing of requirements (initially they might be created manually for testing). Ensure authentication is in place (users can log in and only see their data). If using Django, this might mean setting up Django REST Framework and token auth; if FastAPI, maybe JWT auth.
- **YAML Config Integration:** Write a loader in the backend that reads the YAML file. Use it to populate lookup tables (RequirementType, DocumentType, etc.) or in-memory structures. For now, even loading it into memory and not into DB is fine – the goal is that the system "knows" about the niche specifics. For example, if YAML says a Vendor has X fields, ensure the API for creating an Entity (Vendor) allows those fields. This might require dynamic schema – we can simplify by storing extra fields as JSON in a generic column, or by not enforcing schema at the API layer for those extra fields (just accept a blob of properties). This is a design choice – Weeks 3-4 will involve implementing a strategy for handling custom fields per niche.

- **Testing:** As pieces are built, write basic tests (or at least manual test via Postman) for each API endpoint. For instance, create a vendor via API, retrieve it, update it. This ensures the fundamental backend is solid before adding complexity.
  Milestone by end of Week 4: Core backend up and running with the ability to manage the primary data (for example, the user can create a Vendor and upload an Insurance document via API). The YAML config is loaded and influences the behavior (even if not everything is enforced, the structure is there).

Weeks 5-6: AI Integration & Document Processing

- **Document Upload & Storage:** Implement the endpoint for document upload (if not done in Week 4). When a user uploads, file gets saved to S3 (or locally in dev), a Document record is created. Start integrating the OCR process: after saving file, call OCR (Tesseract or an API) to get text content. Store the raw text temporarily or proceed to parse.
- **LLM Parsing Pipeline:** Integrate with OpenAI API to parse the OCR text. By Week 5, we should have decided on using direct GPT-4 calls vs any finetuning. Implement a function that takes (document_type, extracted_text) and returns a JSON of parsed fields. Use the prompt template from YAML if available. This might involve some experimentation and tweaking the prompt for reliability. Include basic error handling – e.g., if the response is not JSON or some fields missing, perhaps try a second prompt or mark parsing as failed.
- **Linking Data:** When parsed data is obtained, update the corresponding Requirement (or create a new one) with the key date. For example, when an Insurance Cert is uploaded and expiry_date = 2024-12-31 is extracted, the system should either create a new "Insurance Renewal" requirement for that vendor due 2024-12-31, or update an existing one. This logic is niche-specific: since our YAML knows that for Vendor, we have a requirement type tied to that document, we can implement this linkage. This essentially "closes the loop" – document data feeds into the tracking workflow.
- **Notifications Engine:** Set up a simple scheduler. E.g., a daily cron job (could be a Celery beat if using Celery, or a simple thread that checks every N minutes) to scan the database for upcoming or overdue requirements and send out emails. In week 6, we can implement a rudimentary version: pick notifications from the YAML (e.g., 30 days before expiry) and create email content (could use the message template from YAML with Jinja2 substitution). Use a transactional email service API to send the email. For dev/test, we can print to console or use a fake SMTP.

- **Frontend Start (if not earlier):** It would be wise to start the frontend by week 6 if not done alongside backend. At least set up the React project and implement login page and a basic dashboard page. The dashboard can, for now, fetch a list of entities and requirements from the backend and display them. We might use a component library to save time. The reason to start now is to catch any API design issues early (the way the frontend needs data might prompt adjustments in API).
  Milestone by end of Week 6: End-to-end functionality in rudimentary form: You can create an entity (e.g., Vendor) via the frontend or API, upload a document for it, and the system automatically extracts data and schedules a reminder. If we trigger the notification (by faking the date or manually calling the function), an email is sent. Essentially, all main backend pieces (storage, OCR, LLM parse, DB update, email) work in a flow. The frontend is skeletal but can at least display some data.

Weeks 7-8: Frontend Completion and UX Refinement

- **Frontend Forms & Views:** Build out the main UI screens. This includes:
  - Form to add a new Entity (based on the niche, e.g., "Add Vendor" form with fields defined in YAML).
  - List view to show all current Entities (e.g., list of Vendors) and perhaps a detail view for each.
  - Within an Entity detail, show related Requirements (e.g., show "Insurance Policy – due 12/31/2024 – status: OK" for a vendor) and any Documents on file. Allow uploading a new document from this screen.
  - A dashboard or summary page: e.g., a list of all tasks/requirements that are upcoming or overdue across all entities (so the user sees at a glance what needs attention).
  - Notification settings page if we allow customizing who gets emails (could be simple in MVP: always user, so maybe skip this).
  - General polish: add loading spinners, validation messages, etc., to make the app usable and friendly.
- **UX and Design:** In week 7, ideally incorporate any design improvements. If an AI tool or template was planned, apply a CSS framework to make it look professional. Ensure mobile-responsive styling. It doesn't have to be perfect, but navigation and layout should be clean (remember: "simple interface" is key to adoption).

- **AI Feedback UI:** Provide ways for the user to verify or correct AI outputs. For example, after uploading a document and the AI extracts data, in the UI show the extracted values with an edit option. If the AI got something wrong, the user can fix it, and we save the corrected value. This is important for user trust. Implementing this could be as simple as a pop-up "Confirm extracted fields: [Expiry Date: 12/31/2024 ***Correct/Edit***]".
- **Testing (Frontend & Integration):** During week 8, do thorough testing of user flows: sign up, create entity, upload doc, see the result. Test with different sample docs (good to use 2-3 examples to ensure the AI parsing is robust). Fix any bugs in API that surface when used via UI (CORS issues, auth issues, etc.). Also test the notification end-to-end: possibly simulate "time travel" by setting a due date 1 day away and verifying an email is received.
- **Performance Check:** With the app mostly built, consider performance: e.g., ensure large documents don't crash the server (maybe put a size limit), ensure the OCR/AI call is done asynchronously (so the user doesn't wait 30 seconds staring at UI – perhaps implement an async job: upload doc -> show "processing" status -> a background worker completes parse -> update UI). If time permits, integrate a job queue (Celery/RQ) for this. If not, a simpler approach: increase server request timeout and show a spinner with messaging like "Parsing document, this may take up to 20 seconds...".
  Milestone by end of Week 8: The MVP application is feature-complete. A user with no technical knowledge could use the web UI to do the primary use-case: input their data and get automated assistance (AI parsing + reminders) out of it. The app should handle the core scenario reliably. We should also have a visually presentable UI by now.

Week 9: Testing, Bug Fixing, "What Not to Build" Check

- **Bug Bash:** Spend dedicated time to find and fix bugs. This includes edge cases (what if two documents are uploaded for the same requirement? What if the user enters weird data?). Use the application extensively in a staging environment. If possible, involve a friendly beta user or colleague to try it and give feedback.
- **Review "What Not to Build" list:** Ensure that in the rush of development we haven't accidentally over-engineered. For example, if we started building a second niche concurrently and it's jeopardizing quality, consider pausing and focusing back on the first niche. Make sure features that are "nice-to-have" but not necessary are not

delaying launch. If any are partially built (e.g., maybe we started a chatbot interface prematurely), consider hiding or removing them for now to avoid adding complexity.

- **Security & Privacy Sweep:** Check for obvious security issues – e.g., ensure proper auth checks on each API (so one user can't access another's data). Since we handle potentially sensitive documents, ensure files are secured (private S3 buckets or signed URLs, etc.). Also ensure we're not inadvertently storing the AI outputs or API keys in insecure ways.
- **Performance & Scaling Test:** With one user's data it's fine, but test with, say, 50 entities and 100 documents to see if the UI and backend remain responsive. Also test the notification scheduler with many items. This is more of a sanity check; the real scaling concerns come later, but we want to ensure no major inefficiency (like an N+1 DB query issue) in the core design.
- **Finalize Documentation:** Prepare a simple user guide (even if just a doc or in-app help text) explaining how to use the MVP. Also update the README for the repository with setup steps, which will be useful if showing to early users or if open-sourcing anything. If we plan to let others test it, possibly deploy the app on a test URL now. *Milestone by end of Week 9:* The app is stable and ready for limited external testing. All known critical bugs are resolved. We have confidence that we can start onboarding a few users in the next week for feedback.

Week 10: Beta Launch and Feedback

- **Invite Beta Users:** Identify a handful of target users (could be colleagues or a small business owner known to the founder, etc.) who have the pain point we solved. Invite them to try the app. This might involve creating accounts for them and sending them links, or opening up a sign-up with a beta code. The goal is to get real-world usage. If such users are not readily available, alternatively, present the product on a forum like Indie Hackers or a relevant subreddit to find volunteers.
- **Collect Feedback:** Set up a way to capture feedback – perhaps a Google Form, or just schedule calls. Also use analytics (if feasible, integrate a tool like PostHog or even simple logs) to see what parts of the app are used, and where they might get stuck. AI can also assist here: for example, analyze logs or user questions to detect common issues.
- **Minor Improvements:** Based on immediate feedback, fix any UI pain points or clarify any confusing parts of the UX. For example, if beta users don't realize they need to click "Process Document" after upload (if that was a step), maybe streamline it or add instructions. If they report that the AI extracted something

incorrectly, consider if a prompt tweak can fix it. Essentially iterate quickly on any low-hanging fruit that improves usability.

- **Marketing Prep:** Since a public launch is nearing, start preparing marketing materials this week. That includes a simple landing page describing the product (if not already part of the app). Use AI to help write compelling copy explaining the value proposition (e.g., "Never miss a renewal – our AI tracks it all for you"). Maybe prepare a short demo video or screenshots. Also plan the channels to announce (Product Hunt, LinkedIn, etc.) for week 12.
  Milestone by end of Week 10: At least 1–3 real users have used the product and provided feedback. We have testimonials or at least confirmation that the solution actually solves their problem. This validation is critical. We also have initial marketing content ready for launch.

Week 11: Refinement and Hardening

- **Implement Feedback (Iteration):** Focus on any major feedback from beta that hasn't been addressed. If users requested an important feature that is small in scope, consider adding it. For example, perhaps a user said "I really need to export data to Excel" – if that's straightforward (just a CSV download of their tasks), do it in week 11. But be cautious not to start anything big (like a whole new module). Keep within the MVP spirit.
- **Scalability & Monitoring:** Prepare the app for a slightly wider audience. Set up basic monitoring/alerting (so if an error occurs in production, you get notified – could use Sentry for error tracking, or CloudWatch, etc.). Ensure the hosting can handle a surge (e.g., if posting on a forum yields 100 signups, is the email service free tier enough? Is the server sized okay?). We likely stick to free/cheap tiers now, but know the limits. Maybe implement request rate limiting on the API to prevent abuse.
- **Polish & UX:** Do final polishing: better loading states, clear error messages. If time, add small delightful touches (maybe an AI icon indicating when something is AI-generated). Ensure the app looks credible and not "half-done." First impressions matter for new users. Also double-check content: the email templates for notifications – make sure they read well and have correct info (use friendly language, possibly even AI to refine the tone).
- **Documentation & Support:** Create a FAQ section or Help center in the app if needed, addressing likely questions ("How do I choose sample documents?", "What to do if AI gets something wrong?" etc.). Being proactive here will reduce support

burden post-launch. Prepare a simple support plan – since it's just the founder, maybe set up an email like support@yourdomain or a chatbot (could be ChatGPT answering common questions, integrated via API, which would be meta!).
Milestone by end of Week 11: The product is in "release candidate" state. It's polished, all known issues fixed, and ready for official launch. The founder is equipped with materials and support structure for new users.

Week 12: Launch and Beyond

- **Public Launch:** This is the week to go public. Activities include:
    - Posting on Product Hunt or Hacker News (if those are target audiences), including a compelling description and visuals.
    - Announcing on personal/company social media (LinkedIn, Twitter/X), highlighting the problem it solves. Possibly write a short launch blog post.
    - Reaching back out to the initial list of interested users (maybe people who had expressed pain in this area) and inviting them to sign up.
    - Ensuring the landing site is updated with a "Sign Up" or pricing (if any – MVP might be free or invite-only initially to gather users).
- **Collect and Monitor:** Watch the incoming traffic and sign-ups. Keep a close eye on server metrics and error logs now. Also monitor AI usage (OpenAI API usage stats) to ensure it's within budget or if usage spikes, be ready to throttle or temporarily disable something if it risks high cost.
- **Support New Users:** Be ready to answer questions from new signups. Perhaps hold a couple of live onboarding calls or make yourself available in a chat (could use Intercom or even a Discord community for early adopters). Quick, helpful responses will make early users stick around. Use AI to help manage this volume – e.g., draft responses to common questions so you can answer faster.
- **Post-mortem and Plan Next Steps:** As the 90-day sprint completes, take stock of achievements vs. initial plan. Note what went well and what didn't. With the data from launch, decide on the immediate next priorities. For example, if many users sign up in a different niche (say a lot of interest in lease tracking, not just COI), maybe that becomes the next module to build. Or if there's a critical missing feature (e.g., "I want to upload 10 documents at once"), plan to address that. Also evaluate the tech: maybe the OCR is slow, so plan to optimize it or use a better API. This week is about learning from the launch and setting the stage for the **next 90-day cycle** with actual user input.

- **Celebrate:** Ship day should be celebrated! Even as a solo founder, take a moment to acknowledge the accomplishment (perhaps share a post about building a product in 90 days solo with AI help – this itself can drive some marketing, as people love those stories).

Milestone by end of Week 12: The MVP is live and available to the public (or at least broadly to the target market). We have users signing up and a pipeline of feedback and improvements. The project transitions from pure development mode into a combination of maintenance, support, and iterative enhancement based on real-world use.

Throughout this timeline, we remain flexible. If some tasks finish early, we pull in others (e.g., perhaps the frontend took less time thanks to a template, so we start beta a week sooner). Conversely, if something critical takes longer (say, debugging a tricky AI parsing issue), we adjust by postponing non-essentials. The "block" approach above ensures core functionality is done by around week 8, leaving a good buffer for testing and polish which is often where solo projects fall short. By following this calendar, at the 90-day mark we aim to have not just a working product, but one that's been tested and iterated with real input – ready for the journey ahead.

"What Not to Build" (Scope Control)

To maximize our chance of success within 90 days, it's as important to **define what we will NOT build initially** as it is to define what we will. Here is a list of things we will explicitly avoid or defer, to keep the project lean and focused (each item based on either low immediate value, high complexity, or not aligning with the MVP focus):

- **Don't Build All Niches at Once:** We will *not* attempt to fully implement every idea from the analysis PDF in the first version. Trying to launch 5–10 different automation workflows simultaneously would be overwhelming and dilute our efforts. Instead, we pick one core niche (or a cohesive small set) to focus on. This aligns with the principle "Pick a narrow problem…solve specific problems". Other niches will remain on the roadmap and can be added one by one post-MVP.
- **No Perfectionism or Gold-Plating:** We will not spend time on "nice to have" features or perfecting the UI beyond what's necessary for a good user experience. For example, we won't build a complex role-based permission system, custom theming options, or elaborate multi-step wizards in the MVP. If the basic form works, it's enough. As one successful founder's story noted, his first landing page was built in an hour and the MVP was *buggy but functional* – speed and iteration matter more than perfection.

- **Avoid Custom ML Model Development:** We will not invest time in training our own machine learning models from scratch for things like document parsing. That means no developing a custom NLP model or computer vision model in-house during MVP. We'll leverage existing AI services (OpenAI, etc.) which are essentially "model-as-a-service." This avoids a massive R&D time sink and follows the playbook of successful solo AI products that *"leverage existing models via API instead of building your own"*.
- **Don't Build a Native Mobile App Now:** As discussed, the MVP will not include a separate mobile app. No developing in Swift/Java or using Flutter in this timeframe. A polished responsive web app will cover basic mobile usage. Native features like push notifications or offline support will wait. This prevents fragmenting development efforts.
- **No Complex Multi-Tenancy or Admin Overhead (yet):** We won't over-engineer multi-tenant support beyond the basics. For MVP, it's acceptable that the founder manually sets up new client accounts or even that all beta users share the same instance segregated by login. We won't build an elaborate admin portal for managing tenants, billing, feature flags, etc. – those can come when we have paying customers. Similarly, we won't integrate a billing/payments system in the first 90 days (assuming we'll likely run a free beta or pilot). Launching with a free tier or by invitation is fine; monetization can be added once we prove value.
- **No Large-Scale Scalability Features:** We will not prematurely optimize for scale. That means skipping things like multi-region deployments, autoscaling infrastructure, microservices, etc. The MVP on a single decent server or basic scaling will suffice for initial users. If we get lucky and have scale issues, that will be a "good problem" to tackle with real data. Avoiding premature optimization keeps the code simpler and timeline shorter.
- **Skip Edge-Case Automation:** Each niche idea could have endless edge-case automations, but we won't handle every scenario initially. For instance, in lease abstraction, we won't try to handle every possible clause or state law nuance – just general dates and amounts. In permit tracking, we won't integrate with every city's system via API (maybe none at first, relying on user input for status). Essentially, any feature that would require extensive domain-specific logic or data (and thus time) is tabled for later unless it's absolutely core to the value proposition.
- **Minimize Manual Data Entry Alternatives:** The point of the product is automation with AI; however, we need not build full manual overrides or data entry pathways for

every step initially. For example, we won't build a full UI for manually entering every lease clause if AI fails – instead, we handle the few key fields. If the AI misses something non-critical, the MVP might not capture it at all. Over time we can improve either AI or provide manual input options for those fields.

- **No Comprehensive Analytics Dashboard (initially):** While reporting and analytics could be a great feature (e.g., "show me trends of compliance over time"), we won't build extensive reporting tools in MVP. Perhaps a simple count of upcoming tasks is fine. Complex charts, filters, exports – those can be later once data accumulates.
- **Avoid Unproven AI Features:** We will be careful not to implement AI features that sound cool but weren't explicitly identified as solving a pain point. For example, building a chatbot that explains how to fix compliance issues might be neat, but if users didn't ask for it, it's not in MVP. We stick to AI that directly automates what users hate doing (reading documents, tracking dates) rather than futuristic capabilities that might not have immediate ROI.

Sticking to the above "**not** to build" list keeps us on track to deliver the core value quickly. It prevents scope creep which is a major risk in any project, especially one integrating AI (where possibilities can be endless and intriguing). Each item we choose not to build now can be revisited later with a fresh perspective and (hopefully) input from actual users to determine if it's really needed. In short, we focus on the simplest **viable** product: one niche, one core use-case, done really well, and we deliberately say "no" to everything else in the short term.

Glossary of Terms

For clarity, here is a glossary of key terms and acronyms used in this project plan:

- **AI (Artificial Intelligence):** In this context, refers to machine-learning-based services (like GPT-4, computer vision, etc.) that enable the software to perform tasks that normally require human intelligence (e.g., understanding documents, writing text).
- **LLM (Large Language Model):** A type of AI model (like OpenAI's GPT series) trained on vast amounts of text, capable of understanding and generating human-like text. We use LLMs to parse documents and generate summaries or answers in the application.
- **OCR (Optical Character Recognition):** Technology that converts images of text (like scanned papers or PDF files) into actual text data. OCR allows the system to "read" PDF documents by extracting the text for further processing by an LLM or other logic.

- **NLP (Natural Language Processing):** A field of AI focused on interactions with human language. In our plan, NLP techniques (often via LLMs) are used to interpret the text of documents (finding dates, names, etc.) and to allow users to query or get summaries in natural language.
- **Canonical Schema:** The standardized, overarching data model that our system uses to represent information (entities, documents, tasks, etc.) regardless of niche. "Canonical" implies it's the single source of truth structure that all modules adhere to, enabling a unified approach to different domains.
- **YAML (YAML Ain't Markup Language):** A human-readable configuration file format. We use YAML files to define niche-specific configurations (entity types, fields, workflows) that the system can load. It's chosen for its readability and ease of editing.
- **MVP (Minimum Viable Product):** A version of the product with just enough features to be usable by early customers, providing feedback for future development. In this plan, the MVP is what we aim to build in 90 days – delivering the core value (automation of one key process) without all the bells and whistles.
- **SaaS (Software as a Service):** A software delivery model in which users access the application via the internet (usually with a web browser) and typically pay a subscription. Our project is envisioned as a SaaS product, meaning it's hosted online and provided to multiple businesses as a service.
- **Frontend / Backend:** In web development, *frontend* refers to the client-side interface (what runs in the user's browser – the React app, in our case) and *backend* refers to the server-side application and database that power the logic behind the scenes. They communicate via API calls.
- **API (Application Programming Interface):** A set of endpoints that the frontend or third-party services can call to interact with the backend. For example, our backend exposes APIs like POST /api/vendor to add a new vendor. Also, we use external APIs (like OpenAI's) to perform AI tasks.
- **Notification (or Alert):** In this context, an automated message triggered by the system to inform a user of something, typically via email (or SMS). For example, an email reminder that a deadline is approaching. Notifications are a key part of compliance automation (ensuring nothing falls through the cracks).
- **Beta Launch:** A stage where the product is released to a limited audience for testing in real-world scenarios. Beta users understand that the product is in development

and provide feedback and report issues. We plan a beta launch around Week 10 to refine the product before a broader release.

- **No-Code / Low-Code:** Development approaches that require little or no traditional coding, often using visual interfaces or high-level configurations. In our plan, we consider some no-code tools for speed (like using Zapier for integrations, or possibly a no-code front-end) but we do write custom code for core functionality. Low-code aspects might include leveraging frameworks and templates to avoid coding everything from scratch.
- **Domain (Business Domain):** The specific business area or industry we are addressing (e.g., fleet management, construction, real estate leasing). Each domain has niche requirements which we capture via YAML config. "Domain knowledge" refers to understanding the rules and pain points of that industry.

This glossary should help in understanding the specialized terms and abbreviations in the plan. As we implement the project, this list can expand if new terms or acronyms come up (for instance, if we talk about specific regulations like DOT or OSHA, we'd add those). Clear communication is vital in a solo project – even if it's just for oneself or any collaborators, having a shared vocabulary ensures we stay aligned with the plan's intent.