**Backlog: AI-Gedreven Locatie-App voor de Turkse Diaspora (MVP)**

**Epic 1: Foundation & Infrastructure**

*Goal: Establish the core development environment, repository structure, and deployment pipelines for a stable and scalable project foundation.*

**Story E1-S1: Monorepo & Initial Project Setup**

* **Description:** Initialize the monorepo with all top-level directories and configuration files. This includes setting up .gitignore, .editorconfig, licensing, and placeholder directories for Docs/, Infra/, Backend/, and Frontend/. The goal is to have a "hello world" structure that can be built upon.
* **Acceptance Criteria:**
  + Repository can be cloned successfully.
  + All top-level folders (Docs/, Infra/, Backend/, Frontend/, .github/) exist.
  + .gitignore is configured for Python, Node.js, and environment files.
  + A basic README.md file is present with a project description and setup instructions.
* **Notes:** This is a "day zero" task. Keep it simple and focused on structure, not implementation.

**Story E1-S2: Core Backend Infrastructure & Configuration**

* **Description:** Set up the foundational FastAPI application with configuration management, structured logging, and error handling. Create the core application structure without any business logic.
* **Acceptance Criteria:**
  + Running uvicorn app.main:app starts a server without errors.
  + GET /health returns a 200 OK status with a JSON payload {"status": "ok"}.
  + GET /version returns the current project version.
  + Application configuration (e.g., environment) is loaded from a .env file using Pydantic.
  + Logging is implemented in a structured format (JSON) and captures request IDs.
* **Notes:** This story is about making the backend "alive" and observable from the start.

**Story E1-S3: Database Schema Implementation & Migration**

* **Description:** Implement the complete database schema as defined in the data model. Create and run idempotent SQL migration scripts in the Infra/supabase/ folder to create all tables (locations, ai\_logs, tasks, training\_data, category\_icon\_map) with correct columns, data types, constraints, and indexes.
* **Acceptance Criteria:**
  + Running the migration script on a fresh Supabase project creates all tables successfully.
  + All defined columns and constraints (e.g., PRIMARY KEY, UNIQUE on place\_id) are present.
  + Basic indexes exist on foreign keys and frequently queried columns (state, category, next\_check\_at).
  + The backend can establish a connection to the database and perform a simple SELECT 1 query.
* **Notes:** The 0001\_init.sql file is the single source of truth for the database structure.

**Story E1-S4: Frontend Scaffolding and Build Pipeline**

* **Description:** Initialize the chosen frontend (React + Vite) with a basic component structure, build tooling, and a configuration that points to the backend API. The app should display a simple "Hello World" page.
* **Acceptance Criteria:**
  + Running npm run dev starts a local development server.
  + Running npm run build creates a production-ready build without errors.
  + The frontend can read the API base URL from an environment variable (VITE\_API\_BASE\_URL).
  + A basic App.tsx component renders a page title.
* **Notes:** This story is agnostic to the UI library; it's about making the frontend toolchain work.

**Epic 2: Legal Data Discovery & Ingestion**

*Goal: Automatically and legally discover potential Turkish-oriented business candidates from Google APIs and store them in the system.*

**Story E2-S1: Google Places API Integration Service**

* **Description:** Build a robust, quota-efficient service (google\_service.py) that wraps the Google Places API (Nearby Search and Text Search). The service must handle API errors, respect rate limits, and use field masks to request only necessary data.
* **Acceptance Criteria:**
  + Given a latitude, longitude, and radius, the service returns a list of places.
  + Given a text query (e.g., "Turkse bakkerij Rotterdam"), the service returns a list of places.
  + The service implements exponential backoff for retrying failed requests.
  + All API calls are logged for auditing and cost tracking.
* **Notes:** This is a core, reusable service. Focus on stability and efficiency.

**Story E2-S2: DiscoveryBot - Grid-Based Search Scheduler**

* **Description:** Implement the discovery\_bot.py worker. It should read a configuration of target cities (starting with Rotterdam) and categories, execute a grid-based search using the Google Service, and insert new, deduplicated candidates into the locations table with the state CANDIDATE.
* **Acceptance Criteria:**
  + The bot can be run manually from the command line.
  + It processes at least 3 different business categories for Rotterdam.
  + It successfully deduplicates results based on place\_id before insertion.
  + After a run, the locations table contains ~500 new records with state = 'CANDIDATE'.
* **Notes:** Deduplication is critical to avoid wasting API calls and processing power later.

**Story E2-S3: Discovery Task Queue & Automation**

* **Description:** Integrate the DiscoveryBot into the automated task system. Create a mechanism (e.g., a cron job or a call to /pipeline/run) that triggers the discovery process on a scheduled basis (e.g., weekly).
* **Acceptance Criteria:**
  + A task of type DISCOVERY can be created in the tasks table.
  + A scheduled process (e.g., Render Cron) successfully triggers the DiscoveryBot.
  + The DiscoveryBot run is logged, and its status (success/failure) is recorded.
* **Notes:** This moves the system from manual execution to a fully automated pipeline.

**Epic 3: AI-Powered Classification & Verification**

*Goal: Use AI to intelligently filter, categorize, and enrich candidate businesses, moving them to a verified state.*

**Story E3-S1: OpenAI Service with Structured JSON Output**

* **Description:** Build the openai\_service.py that handles communication with the OpenAI API. The service must be able to enforce a JSON schema on the AI's output, handle retries, and log all interactions for cost and quality analysis in the ai\_logs table.
* **Acceptance Criteria:**
  + The service can send a prompt and receive a valid JSON response conforming to a predefined Pydantic model.
  + If the AI returns invalid JSON, the service retries the request (up to a limit).
  + Every AI call, including input, output, and model used, is recorded in the ai\_logs table.
* **Notes:** Reliability of the AI interaction is key. Invalid JSON should be the exception, not the rule.

**Story E3-S2: AI Classification - Relevance & Categorization**

* **Description:** Implement the classify AI task. It should take a candidate's name, address, and type and determine: a) if it's relevant to the Turkish diaspora (keep/ignore), and b) its primary category (e.g., bakery, restaurant).
* **Acceptance Criteria:**
  + The classification prompt includes few-shot examples with clear Dutch/Turkish patterns.
  + On a test set of 100 manually labeled candidates, the classification precision is ≥ 90%.
  + The output is successfully validated against a JSON schema.
  + The candidate's category and confidence\_score are updated in the database based on the result.
* **Notes:** This is the first major quality gate. The prompt engineering here is critical.

**Story E3-S3: VerificationBot - Data Enrichment & Verification**

* **Description:** Implement the verification\_bot.py worker. For high-confidence candidates, it should call the Google Place Details API to fetch rich data like website, rating, user\_ratings\_total, and precise business\_status. It then uses an AI enrichment step to consolidate this information and update the candidate's state to VERIFIED or SUSPENDED.
* **Acceptance Criteria:**
  + The bot fetches Place Details using a field mask to be quota-efficient.
  + The AI enrichment step successfully extracts and validates the website and other fields.
  + Records with business\_status like CLOSED\_PERMANENTLY are moved to SUSPENDED.
  + Verified records have populated rating, website, and last\_verified\_at fields.
* **Notes:** This step adds the most value for the end-user by providing rich, trustworthy data.

**Story E3-S4: Confidence Scoring System**

* **Description:** Develop a heuristic algorithm that calculates a confidence\_score (0-1) for each location. The score should combine factors like the source reliability, data consistency, number of reviews, and data freshness.
* **Acceptance Criteria:**
  + A confidence\_score is calculated for every record during the verification process.
  + The score is stored in the database and is used to prioritize records in the frontend and for monitoring.
  + The scoring logic is documented and can be tuned based on real-world performance.
* **Notes:** This score is crucial for automating data quality decisions without human intervention.

**Epic 4: Frontend Map & List Application**

*Goal: Build a fast, intuitive, and mobile-friendly web application for end-users to discover Turkish-oriented businesses.*

**Story E4-S1: Core Map View with Leaflet**

* **Description:** Implement the MapView component using Leaflet. It should display a map of the Netherlands, centered on a default location or the user's position, with markers for verified locations.
* **Acceptance Criteria:**
  + The map loads correctly with OpenStreetMap or a similar tile layer.
  + Markers are displayed on the map for locations fetched from the GET /api/v1/locations endpoint.
  + Clicking a marker displays a popup with the location's name and category.
  + The map performance is smooth with up to 200 markers.
* **Notes:** This is the primary visual interface for the app.

**Story E4-S2: Location List & Filtering Components**

* **Description:** Build the LocationList and LocationCard components. The list should display locations, be sortable by distance and rating, and filterable by category. The card should show all key information.
* **Acceptance Criteria:**
  + The list displays location name, address, category, rating, and status badge.
  + Users can filter the list by selecting one or more categories.
  + Users can sort the list by "Distance" or "Rating".
  + The list and map are synchronized; clicking a list item highlights the corresponding marker on the map.
* **Notes:** This provides an alternative, accessible way to browse the data.

**Story E4-S3: Status Badges & Business Logic**

* **Description:** Implement the Badge component and the logic to determine which status badge to show ("Nieuw", "Binnenkort open", "Tijdelijk gesloten", "Gesloten") based on the state, business\_status, and is\_probable\_not\_open\_yet fields.
* **Acceptance Criteria:**
  + A "New" badge is shown for records verified within the last 7 days.
  + A "Coming Soon" badge is shown if is\_probable\_not\_open\_yet is true.
  + A "Temporarily Closed" badge is shown for the corresponding business\_status.
  + The badge logic is consistent between the MapView popups and the LocationList.
* **Notes:** These badges are a key UX feature for establishing trust and data freshness.

**Story E4-S4: Geolocation & Language Internationalization**

* **Description:** Implement a useUserPosition hook to request and use the browser's geolocation. Also, set up an i18n system to toggle the UI language between Dutch (nl) and Turkish (tr).
* **Acceptance Criteria:**
  + The app asks for user location permission on first load (with a fallback to a default center).
  + The map and list automatically update to show locations near the user.
  + A language toggle button switches all UI text (buttons, labels, badges) between Dutch and Turkish.
  + All static text is loaded from nl.json and tr.json files.
* **Notes:** These features directly cater to the core user base and improve accessibility.

**Epic 5: Continuous Monitoring & Data Freshness**

*Goal: Ensure the data remains accurate and up-to-date through automated monitoring and re-verification processes.*

**Story E5-S1: MonitorBot & Freshness Policy Engine**

* **Description:** Implement the monitor\_bot.py worker. It should periodically select records where next\_check\_at is in the past, based on the defined Freshness Policy. It then re-triggers the verification process for these records.
* **Acceptance Criteria:**
  + The bot correctly calculates next\_check\_at for new records based on their confidence and status.
  + The bot processes records in batches (e.g., MONITOR\_MAX\_PER\_RUN=200).
  + After a run, no record in the database has a next\_check\_at older than 90 days.
  + Changes to a location's business\_status (e.g., from operational to closed) are detected and updated.
* **Notes:** This is the "heartbeat" of the application, ensuring long-term data quality.

**Story E5-S2: Admin Dashboard for System Health**

* **Description:** Create a simple, private admin dashboard (or enhance existing admin endpoints) that displays key system metrics. This should include counts of locations per state, the status of the last bot runs, and a list of recent errors.
* **Acceptance Criteria:**
  + Accessing the admin dashboard shows a summary of locations grouped by state.
  + The dashboard displays the timestamp and status (success/failure) of the last run for each bot.
  + The top 5 most recent task failures are visible.
* **Notes:** This gives the product owner and developers at-a-glance insight into system health.

**Epic 6: Admin CMS & Human-in-the-Loop**

*Goal: Provide tools for administrators to manually correct data, add gold records, and manage the system.*

**Story E6-S1: Manual Location Management & Gold Records**

* **Description:** Build admin-only API endpoints and a simple UI (or use a tool like Postgres Studio) to allow admins to manually create, edit, and correct location records. These manually added records should be flagged as "gold" in the training\_data table.
* **Acceptance Criteria:**
  + An admin can add a new location with all core fields, which is immediately set to VERIFIED.
  + An admin can edit any field of an existing location.
  + All manual actions are logged for audit purposes.
  + Manually added/verified records are stored as gold records for the LearningBot.
* **Notes:** This is the "human-in-the-loop" mechanism for continuous improvement.

**Story E6-S2: Data Export Functionality**

* **Description:** Implement an admin endpoint GET /admin/export-locations that exports all location data to a CSV file for external analysis or auditing.
* **Acceptance Criteria:**
  + The endpoint returns a CSV file with all columns from the locations table.
  + The export includes all records, regardless of state.
  + The file is generated server-side and downloads correctly.
* **Notes:** This is a simple but powerful feature for data backup and offline analysis.

**Epic 7: Observability, Security & Hardening**

*Goal: Make the system reliable, secure, and production-ready with monitoring, alerts, and compliance checks.*

**Story E7-S1: Comprehensive Metrics & Alerting**

* **Description:** Implement a metrics collection system that tracks key KPIs: new candidates per week, conversion rate to VERIFIED, task error rates, and API latency. Set up basic alerting (e.g., via email) for critical failures like repeated task errors or API quota warnings (429s).
* **Acceptance Criteria:**
  + A metrics dashboard (e.g., a simple log-based dashboard or a Prometheus setup) displays at least 5 core metrics.
  + An alert is triggered when the task failure rate exceeds 10% in a 1-hour period.
  + An alert is triggered when the Google API returns a burst of 429 status codes.
* **Notes:** You can't manage what you can't measure. This is essential for production.

**Story E7-S2: Security Hardening & Rate Limiting**

* **Description:** Implement security best practices. This includes adding rate limiting to all public API endpoints, ensuring no API keys are exposed to the frontend, and writing a basic privacy policy and cookie notice for the frontend.
* **Acceptance Criteria:**
  + The GET /api/v1/locations endpoint has a rate limit (e.g., 60 requests per minute).
  + A security scan confirms that no secrets (Google/OpenAI keys) are bundled in the frontend.
  + The live frontend application displays a minimal, compliant cookie notice.
  + A privacy-&-legal.md document is published in the Docs/ folder.
* **Notes:** This is non-functional but critical for public launch.

**Epic 8: Pilot Deployment & Roll-out**

*Goal: Prepare the MVP for its first real-world test, gather feedback, and plan for the next phase.*

**Story E8-S1: End-to-End Pilot Deployment**

* **Description:** Deploy the entire integrated system (Backend, Frontend, Database, Cron Jobs) to a production environment. Execute a full, automated cycle from discovery to verification for Rotterdam and ensure the public-facing app is stable and performant.
* **Acceptance Criteria:**
  + The public URL for the frontend is live and accessible.
  + The application has ≥200 VERIFIED locations in Rotterdam with ratings and websites.
  + All automated bots (Discovery, Verification, Monitor) are running on schedule without human intervention.
  + The application loads in under 2 seconds on a mobile 3G connection.
* **Notes:** This is the "go-live" story. The success criteria for the MVP are met here.

**Story E8-S2: Pilot Review & V2 Backlog Creation**

* **Description:** After the pilot has run for 2 weeks, compile a pilot report. Analyze the KPIs (data coverage, AI precision, costs). Conduct user interviews and synthesize all feedback into a prioritized backlog for Version 2.
* **Acceptance Criteria:**
  + A "Pilot Report" document is completed, covering quality, coverage, costs, and user feedback.
  + A prioritized list of features, bugs, and improvements for V2 is created in the project management tool.
  + A formal "Go/No-Go" decision for further investment in the project is made by the Product Owner.
* **Notes:** This story ensures the project is data-driven and user-focused for its next phase.