Of course. A successful project is built on a foundation of clarity, precision, and a shared understanding of the work ahead. This action plan is designed to provide exactly that. It breaks down the entire MVP development process into a logical sequence of self-contained, verifiable steps.

Each step represents a distinct milestone. This plan should be treated as the single source of truth for the development team, ensuring we build the right features, in the right order, to the highest standard.

**Step 1: Foundational Architecture & Project Setup**

This initial step is the most critical. We are laying the concrete foundation for the entire application. The focus is on establishing a clean, scalable architecture with a clear separation of concerns. This ensures that future features can be added efficiently without requiring major refactoring. We will create the core projects, define the database context, and model our primary data entities.

**Task Breakdown**

* **SubTask 1: Initialize Solution and Core Projects**
  + **Description:** Create the main solution file (.sln). Inside, initialize a .NET Standard class library project named Core. This library will contain all shared business logic, data models, and services, making it platform-agnostic.
  + **/relative/path/of/changed/file:** /src/Core/Core.csproj
  + **Operation being done:** Create
* **SubTask 2: Initialize Platform-Specific UI Projects**
  + **Description:** Within the same solution, initialize a C# WPF project named WindowsUI and a SwiftUI project named MacUI. Configure both projects to have a project reference to the Core library.
  + **/relative/path/of/changed/file:** /src/WindowsUI/WindowsUI.csproj and /src/MacUI/MacUI.xcodeproj
  + **Operation being done:** Create
* **SubTask 3: Integrate SQLite and Entity Framework Core**
  + **Description:** Add the necessary NuGet packages for SQLite and Entity Framework Core (e.g., Microsoft.EntityFrameworkCore.Sqlite) to the Core project. Create the KitchenDbContext.cs file that defines the connection and the database tables (DbSet).
  + **/relative/path/of/changed/file:** /src/Core/Data/KitchenDbContext.cs
  + **Operation being done:** Create
* **SubTask 4: Define Core Data Models and Initial Migration**
  + **Description:** Create the plain C# object models for Item, Category, User, and UsageHistory within the Core project, including all fields, types, and relationships (UUIDs for IDs, foreign keys). Generate the initial EF Core migration to create the database schema from these models.
  + **/relative/path/of/changed/file:** /src/Core/Data/Models/ (all model files) and /src/Core/Data/Migrations/ (initial migration file)
  + **Operation being done:** Create

**Other Notes On Step 1: Foundational Architecture & Project Setup**

* **Blocked by:** N/A. This is the starting point.
* **Manual Tasks:** Ensure the correct .NET SDK and platform development tools (Visual Studio 2022+ with WPF workload, Xcode 15+) are installed. Verify that the initial migration can be successfully applied to create a local .db file.

**Step 2: Implement User Authentication & Role Management**

With the foundation in place, our first functional task is to secure the application. This step focuses on building the complete user management and authentication flow. We will implement the logic for logging in, hashing passwords, and creating a session state that will be used by all subsequent features to authorize actions.

**Task Breakdown**

* **SubTask 1: Build the Authentication Service**
  + **Description:** In the Core library, create an AuthenticationService responsible for user login, password hashing (using Argon2 or bcrypt), and verification. This service will interact with the Users table via a repository.
  + **/relative/path/of/changed/file:** /src/Core/BusinessLogic/Services/AuthenticationService.cs
  + **Operation being done:** Create
* **SubTask 2: Create the Login UI and ViewModel**
  + **Description:** In the WindowsUI and MacUI projects, build the login screen UI (username/password fields, login button). Create a corresponding LoginViewModel that captures user input and uses the AuthenticationService to validate credentials.
  + **/relative/path/of/changed/file:** /src/WindowsUI/Views/LoginWindow.xaml and /src/WindowsUI/ViewModels/LoginViewModel.cs
  + **Operation being done:** Create
* **SubTask 3: Implement Session Management**
  + **Description:** Upon successful login, create a simple static session manager or application-level state object to store the current user's ID and Role (Admin, Manager, Staff). This state must be globally accessible throughout the application's lifecycle.
  + **/relative/path/of/changed/file:** /src/Core/BusinessLogic/Services/UserSession.cs
  + **Operation being done:** Create
* **SubTask 4: Build Admin-Only User Management UI**
  + **Description:** Create the UI and corresponding ViewModel for an Admin-only user management screen. This view will allow Admins to perform full CRUD operations on the User entity. The view should only be accessible if UserSession.Role is "Admin".
  + **/relative/path/of/changed/file:** /src/WindowsUI/Views/UserManagementView.xaml
  + **Operation being done:** Create

**Other Notes On Step 2: Implement User Authentication & Role Management**

* **Blocked by:** Step 1 (Foundation and User model).
* **Manual Tasks:** Manually add a default 'admin' user to the database (or create seeding logic) to allow for the first login. Test that non-admin users cannot access the user management screen.

**Step 3: Implement Category and Item CRUD Operations**

This step delivers the primary value of the application: managing the inventory. We will build the core functionality that allows users to add, view, modify, and delete categories and items, with all actions respecting the user's role defined in the previous step.

**Task Breakdown**

* **SubTask 1: Implement Category CRUD**
  + **Description:** Create the service, repository, and Admin-only UI to allow full CRUD operations on Category entities. Staff and Manager roles should not be able to see or interact with this UI.
  + **/relative/path/of/changed/file:** /src/Core/BusinessLogic/Services/CategoryService.cs and /src/WindowsUI/Views/CategoryManagementView.xaml
  + **Operation being done:** Create
* **SubTask 2: Implement Item Create/Update Operations**
  + **Description:** Build the InventoryService in the Core library. Create a reusable form/modal UI (ItemEditView) for adding a new item and editing an existing one. The form should include validation (e.g., name not empty, quantity non-negative).
  + **/relative/path/of/changed/file:** /src/Core/BusinessLogic/Services/InventoryService.cs and /src/WindowsUI/Views/ItemEditView.xaml
  + **Operation being done:** Create
* **SubTask 3: Implement Item Read (List, Search, Filter)**
  + **Description:** Create the main inventory list view. This view will display all non-deleted items in a data grid. Implement real-time search (on name) and filtering (on category) functionality within the InventoryViewModel.
  + **/relative/path/of/changed/file:** /src/WindowsUI/Views/InventoryView.xaml and /src/WindowsUI/ViewModels/InventoryViewModel.cs
  + **Operation being done:** Create
* **SubTask 4: Implement Item Delete Operation & History Tracking**
  + **Description:** Add a "Delete" button to the inventory list. The action will perform a soft delete (IsDeleted = true). Crucially, every quantity change (from create, update, or delete) must be recorded in the UsageHistory table by the InventoryService.
  + **/relative/path/of/changed/file:** /src/Core/BusinessLogic/Services/InventoryService.cs
  + **Operation being done:** Update

**Other Notes On Step 3: Implement Category and Item CRUD Operations**

* **Blocked by:** Step 2 (User roles must be available to authorize actions).
* **Manual Tasks:** Test the feature using all three user roles (Admin, Manager, Staff) to ensure permissions are correctly enforced for every CRUD action.

**Step 4: Implement Notifications & Rule-Based Suggestions**

Now that the core data can be managed, this step makes the application proactive. We will implement the logic to monitor inventory levels and provide intelligent, rule-based suggestions to the user, helping them prevent stockouts.

**Task Breakdown**

* **SubTask 1: Create the Suggestion Service**
  + **Description:** In the Core library, create a SuggestionService. This service will contain the logic to check for items below their LowStockThreshold and to calculate the average daily usage from the UsageHistory table for forecasting.
  + **/relative/path/of/changed/file:** /src/Core/BusinessLogic/Services/SuggestionService.cs
  + **Operation being done:** Create
* **SubTask 2: Implement Background Check**
  + **Description:** On application startup, and perhaps on a timer, trigger the SuggestionService to run its checks. This should be a lightweight background task that does not freeze the UI.
  + **/relative/path/of/changed/file:** /src/WindowsUI/App.xaml.cs
  + **Operation being done:** Update
* **SubTask 3: Display Notifications and Suggestions in UI**
  + **Description:** Create a dedicated area in the main UI (e.g., a "Needs Attention" panel or a notification icon) to display the generated suggestions. Each suggestion should be clear, actionable, and provide a reason (e.g., "Low Stock" or "High Usage").
  + **/relative/path/of/changed/file:** /src/WindowsUI/Views/DashboardView.xaml or /src/WindowsUI/Views/MainWindow.xaml
  + **Operation being done:** Create
* **SubTask 4: Add Visual Alerts to Inventory List**
  + **Description:** In the main inventory list, apply visual styling (e.g., a yellow or red background color) to rows corresponding to items that have an active reorder suggestion or are nearing their expiration date.
  + **/relative/path/of/changed/file:** /src/WindowsUI/Views/InventoryView.xaml
  + **Operation being done:** Update

**Other Notes On Step 4: Implement Notifications & Rule-Based Suggestions**

* **Blocked by:** Step 3 (Requires item data and usage history to function).
* **Manual Tasks:** Populate the UsageHistory table with varied data to test the accuracy of the average usage calculation and suggestion logic under different scenarios.

**Step 5: Finalizing, Testing, and Debugging**

This final step is about ensuring quality and stability. We will rigorously test every feature, both in isolation (unit tests) and as part of the whole application (integration tests). The goal is to identify and fix bugs, polish the user experience, and prepare a release candidate.

**Task Breakdown**

* **SubTask 1: Write Unit Tests for Core Logic**
  + **Description:** Create a new test project (Core.Tests). Write unit tests for all public methods in the services (AuthenticationService, InventoryService, SuggestionService), mocking the database repositories to test the business logic in isolation.
  + **/relative/path/of/changed/file:** /tests/Core.Tests/ (all test files)
  + **Operation being done:** Create
* **SubTask 2: Write Integration Tests**
  + **Description:** Write integration tests that interact with an in-memory or temporary test database. These tests will verify the full flow from the service layer to the database, ensuring EF Core mappings and queries are correct.
  + **/relative/path/of/changed/file:** /tests/Core.IntegrationTests/ (all test files)
  + **Operation being done:** Create
* **SubTask 3: Full Manual User Acceptance Testing (UAT)**
  + **Description:** Perform a complete, end-to-end manual test of the application, following predefined test cases for all user stories and roles. Document all bugs, UI/UX issues, and unexpected behavior.
  + **/relative/path/of/changed/file:** /docs/TestCases.md
  + **Operation being done:** Create/Update
* **SubTask 4: Bug Fixing and Performance Polishing**
  + **Description:** Address all critical and major bugs identified during testing. Profile the application to identify and fix any performance bottlenecks, particularly around database queries and UI rendering on the inventory list.
  + **/relative/path/of/changed/file:** (Various files across the solution)
  + **Operation being done:** Update

**Other Notes On Step 5: Finalizing, Testing, and Debugging**

* **Blocked by:** All core features (Steps 2, 3, 4) must be code-complete.
* **Manual Tasks:** The entire team should participate in UAT to gather diverse feedback. Focus on edge cases (e.g., what happens if you try to add an item with a negative quantity? What if the database file is deleted?).