Project Architecture & Thought Process

System Overview

The project was structured around a **clear separation of concerns**, with a focus on scalability, maintainability, and explicit ownership. Core gameplay systems (movement, interaction, inventory) are **state- and data-driven**, while the UI layer remains fully reactive.

The **Player** is composed of focused subsystems: input handling, a state machine, interaction detection, and inventory ownership. Input is captured once and interpreted by the state machine, preventing direct coupling between input and behavior. Actions such as movement, attacking, and interacting are mutually exclusive and enforced through explicit states.

World interaction is implemented through a clean `IInteractable` contract. Interactable objects expose intent (interaction text, anchor point, busy state) without owning UI or input logic. The player detects nearby interactables, selects a valid target deterministically, and drives a single world-space prompt UI that follows the selected object.

The inventory system follows an **MVP architecture** to ensure gameplay logic is independent from UI. The **Model** owns all inventory data and rules and is used directly by gameplay systems. The **Presenter** synchronizes model changes to the **View**, which is a passive Canvas-only layer. As a result, the inventory can function and be tested without any UI dependency. Drag-and-drop and click interactions are handled entirely in the UI layer via EventSystem pointer events, remaining compliant with the Unity Input System.

Audio is implemented as a lightweight service using ScriptableObject-based sound definitions. Sounds are triggered through gameplay or animation events, ensuring consistent timing without frame-based playback. Save and load functionality is handled via explicit DTOs and stable item IDs, keeping persistence decoupled from Unity objects and UI.

Thought Process & Personal Assessment

Throughout the project, I intentionally avoided common prototyping shortcuts, such as tightly coupling gameplay logic to MonoBehaviours or UI-driven data manipulation, in favor of explicit responsibilities and clear system boundaries. While the architecture is designed to scale, some production-level features were intentionally omitted due to time constraints.

These include a full Dependency Injection solution, object pooling for UI and world objects, and a more robust save system with versioning or platform-specific backends. The current structure was designed so these improvements could be added without refactoring core systems.

Overall, I believe the solution accurately reflects how I approach real-world gameplay architecture problems under production constraints.