**DESIGN DOCUMENT: INVENTORY MANAGEMENT SYSTEM WITH NLP INTERFACE**

**1. OVERVIEW**

This document outlines the design approach for the Inventory Management System with Natural Language Interface. The system enables users to manage inventory through natural language commands, bridging the gap between technical APIs and human communication.

**2. KEY DESIGN**

**2.1 Two-Component Architecture**

Decision: Split the system into two separate components (Inventory Service and MCP Server)

Reasoning:

- Clear separation of concerns between inventory management and natural language processing

- Independent scalability of each component

- Follows the task requirements explicitly

- Allows for focused testing and development of each component

- Enables future replacement or enhancement of either component without affecting the other

**2.2 Serverless Inventory Service**

Decision: Implement the Inventory Service as an AWS Lambda function with API Gateway

Reasoning:

- Minimal operational overhead - no server management required

- Automatic scaling to handle variable loads

- Cost efficiency with pay-per-use model

- Quick deployment with AWS SAM

- Focuses development effort on business logic rather than infrastructure

**2.3 In-Memory Storage with Singleton Pattern**

Decision: Use a Singleton class to manage in-memory inventory state

Reasoning:

- Satisfies the project requirement for in-memory storage

- Provides proper encapsulation of data and operations

- Follows object-oriented design principles

- Enables easier unit testing through dependency injection

- Maintains state between Lambda invocations (when container stays warm)

**2.4 Hybrid NLP Approach**

Decision: Combine regex pattern matching with OpenAI's GPT for natural language processing

Reasoning:

- Faster response times for common queries using regex

- More cost-efficient by avoiding API calls for simple cases

- Better flexibility for handling complex or unusual phrasings with GPT

- Improved reliability with fallback mechanisms

- Adaptable to different query patterns without extensive regex updates

**2.5 OpenAPI Specification Integration**

Decision: Use OpenAPI specification to define and document the Inventory API

Reasoning:

- Self-documenting API for better developer experience

- Enables dynamic discovery of API capabilities by the MCP Server

- Provides schema validation for requests and responses

- Creates a contract between services that's language-agnostic

- Follows industry standards for API design

**2.6 FastAPI for MCP Server**

Decision: Implement the MCP Server using FastAPI

Reasoning:

- High performance with async support

- Automatic OpenAPI documentation generation

- Built-in data validation with Pydantic

- Modern Python syntax with type hints

- Developer-friendly with clear error messages

**2.7 Modular Code Organization**

Decision: Organize code into focused modules with specific responsibilities

Reasoning:

- Improves maintainability by separating concerns

- Enables easier testing of individual components

- Facilitates collaborative development

- Makes the codebase more navigable

- Allows for easier extension and enhancement

**3. CHOSEN APPROACH: SERVERLESS ARCHITECTURE WITH HYBRID NLP**

**3.1 Architecture Overview**

I implemented a two-component architecture:

1. Inventory Service

- AWS Lambda serverless function

- API Gateway for REST interface

- Singleton pattern for in-memory storage

- AWS SAM for deployment

2. MCP (Model Control Plane) Server

- FastAPI web application

- Hybrid NLP system (regex + OpenAI)

- OpenAPI specification parsing

- Modular design with clear separation of concerns

**3.2 Data Flow**

1. User submits natural language query (e.g., "add 5 shirts")

2. MCP Server processes query using pattern matching or OpenAI

3. MCP Server converts query to structured API call

4. Inventory Service processes the API call and returns updated inventory

5. MCP Server formats the response in natural language

6. User receives a friendly confirmation

**4. ALTERNATIVE APPROACHES CONSIDERED**

**4.1 Simple Monolithic Web Application**

A simpler alternative would be to combine both components into a single web application.

Architecture:

- Single Web Server: Flask or FastAPI application

- Unified Codebase: Inventory management and NLP processing in one application

- Direct Integration: No need for API calls between components

Advantages:

- Simplicity: Easier to develop and deploy as a single unit

- Lower Latency: No network calls between components

- Shared Memory: Direct access to inventory data from NLP code

Disadvantages:

- Limited Scalability: Cannot scale components independently

- Higher Coupling: Changes to one part affect the entire application

- Reduced Flexibility: More difficult to replace or enhance individual components

**4.2 Fully Client-Side Approach**

Another simpler alternative would be to handle everything in the browser with a JavaScript application.

Architecture:

- Static Website: HTML, CSS, and JavaScript hosted on S3 or similar

- Browser-Based Processing: JavaScript handles inventory logic

- Direct OpenAI Integration: Browser makes calls directly to OpenAI API

Advantages:

- Zero Backend Infrastructure: No server-side components to manage

- Simplified Deployment: Just static files to host

- Instant Updates: Changes immediately reflected in the UI

Disadvantages:

- Security Concerns: API keys exposed in client-side code

- Limited Persistence: Data only stored in browser (localStorage)

- Performance Issues: Limited processing power for complex operations

- Cost Control Challenges: Harder to manage and limit API usage

**5. JUSTIFICATION OF APPROACH**

I selected the Serverless Architecture with Hybrid NLP approach because:

1. Requirement Alignment: It directly satisfies the project requirements for separate components and in-memory storage.

2. Scalability: Each component can scale independently based on its specific load.

3. Cost Efficiency: The serverless model ensures we only pay for what we use.

4. Development Focus: Allows us to focus on implementing business logic without infrastructure concerns.

5. Future Extensibility: The modular design makes it easier to add features or replace components.

While the alternative approaches might be simpler for initial development, they would not satisfy the project requirements as effectively and would create limitations for future enhancements.

**6. CONCLUSION**

My design decisions were driven by a balance of requirement adherence, development efficiency, and future flexibility. The serverless architecture with hybrid NLP approach provides the optimal solution for this inventory management system, meeting all project requirements while providing a scalable and maintainable foundation.

The simpler alternatives would require compromises in either functionality, scalability, or requirement adherence that were not acceptable for this project. However, elements of these alternatives could be considered for different use cases or if requirements change in the future.