Tapster: Design Brief

A personal bartender and recipe management agent.

PEAS Analysis

Performance Measures

- **Consistency**: Saved recipes must be returned exactly as stored (same ratios, units, ingredients)
- **Predictability**: New recipes must not deviate significantly from established cocktail standards and conventions
- **Accuracy**: Recipe information retrieved from searches must be faithfully represented when saved
- **Helpfulness**: Successfully finding relevant recipes that match user requests and providing useful cocktail information
- **Safety**: Dangerous recipes (i.e. immediately poisonous, obviously alcohol is itself dangerous) are refused

Environment

Observability: Partially Observable - Tapster cannot access the entire internet or exhaustively search all database contents. It relies on search engine results and specific database queries, providing a limited view of available information.

Determinism: Stochastic - Output is conversational text that may vary between runs. Search results from The Cocktail DB may also change over time, and LLM responses introduce non-deterministic elements.

Episodes vs Sequence: Sequential - Actions have lasting effects. Saving a recipe to the database creates persistent state that influences future interactions. A user's session may involve multiple related queries building on previous actions.

Dynamics: Static - The database remains unchanged during individual operations, and web search results are stable for the duration of a single query. The environment doesn't change while the agent is processing.

State Space: Discrete - Actions occur only when prompted by user input. State transitions happen at distinct moments rather than continuously.

Agent Count: Single Agent - One Tapster agent with access to multiple tools operating independently.

Actuators/Actions

- **Send conversational responses** to user queries in natural language
- **Search the web** using The Cocktail DB API for cocktail recipes and information
- **Store recipes** to SQLite database with proper normalization
- **Query database** for saved recipes and ingredient information
- **Create new recipes** from user specifications with validation

Sensors/Percepts

- **User queries** in natural language requesting recipes, storage, or information
- **Search results** from The Cocktail DB containing recipe information, ingredients, and instructions

- **Database query results** returning cocktail and ingredient data
- **Recipe validation feedback** ensuring ingredients and proportions are reasonable for cocktails

Agent Design

Agent Type: Hybrid Architecture

Tapster employs a hybrid approach combining:

- **Reflex behavior** for simple queries like "show me saved Old Fashioned recipes" that trigger direct database lookups
- **Deliberative planning** for complex requests like "find and save a whiskey sour recipe" requiring multi-step reasoning: search → evaluate results → extract recipe data → validate → store

Memory and State Management

- **No persistent agent memory** between sessions - each interaction starts fresh
- **Database serves as external memory** for storing and retrieving recipes
- **Conversation context** maintained within single sessions for follow-up questions

Prompting Strategy

- **System prompt** establishes Tapster's persona as a knowledgeable, friendly bartender
- **Tool use prompts** provide clear instructions for when and how to use tools
- **Validation prompts** include safety checks for ingredient appropriateness and recipe reasonableness
- **Output formatting** guidelines ensure consistent recipe presentation

Tool Architecture

Get Cocktail Tool (`GetCocktail`)

- **Input**: Cocktail name
- **Output**: Cocktail recipe markdown

Save Cocktail Tool (`SaveCocktail`)

- **Input**: Cocktail JSON string
- **Output**: A boolean indicating success or failure saving the

Recipe Validation Tool (`RecipeValidator`)

- **Input**: Cocktail recipe
- **Output**: Boolean indicating ingredients are safe for human consumption

Evaluation Plan

Test Scenarios

1. **Search and Save**: "Find me a whiskey sour recipe and save it"
    - Success: Recipe found, properly extracted, and stored in database
    - Metrics: Search relevance, data extraction accuracy, storage success
2. **Retrieval**: "Show me a whiskey sour recipe"
    - Success: Saved recipe returned with complete information
    - Metrics: Retrieval completeness, formatting consistency

3. **Custom Creation**: "Create a new recipe called an Old Fashioned that's made with 2oz of Bourbon, 1/4oz of Demerara Syrup, and 1/8oz of Aromatic Bitters"
    - Success: Recipe created with proper ingredient parsing and storage
    - Metrics: Ingredient extraction accuracy, quantity parsing, validation effectiveness
4. **Safety Validation**: "Is a cocktail safe that contains arsenic safe?"
    - Success: A message indicates that such a cocktail is unsafe
    - Metrics: User safety

Success Criteria

- **Task Completion Rate**: ≥80% of test scenarios completed successfully
- **Data Integrity**: 100% consistency between stored and retrieved recipes
- **Response Quality**: Conversational responses rated as helpful and bartender-appropriate
- **Error Handling**: Graceful failure modes with informative error messages

Metrics Collection

- Success/failure rates for each test scenario
- Response time for different operation types
- Database integrity checks after operations
- Qualitative assessment of conversational quality

Risk Assessment

Failure Modes

- **Recipe Safety**: Filtering dangerous ingredient combinations or inedible substances
- **Data Quality**: Handling incomplete or malformed search results
- **Database Corruption**: Preventing invalid data from breaking the recipe storage system
- **Search Failures**: Graceful handling when no relevant recipes are found

Mitigation Strategies

- **Ingredient Validation**: Maintain a disallowlist of unsafe cocktail ingredients
- **Data Sanitization**: Clean and validate all inputs before database operations
- **Fallback Responses**: Helpful error messages when operations fail

Implementation Notes

Database Schema

Primary implementation will focus on the `cocktails` table for simplicity:

```
CREATE TABLE cocktails (
    cocktail_id INTEGER PRIMARY KEY,
    name TEXT UNIQUE NOT NULL,
    instructions TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Environment Variables

- `GEMINI_API_KEY`: Google Gemini API access

Dependencies

- `smolagents`: Agent framework and tool integration
- `sqlite3`: Database operations
- `requests`: Web search API calls