Tapster: Design Brief

A personal bartender and recipe management agent designed to assist users with discovering, creating, and managing cocktail recipes, while also providing helpful information regarding drinks.

PEAS Analysis

Performance measure(s)

- **Consistency**: Saved recipes must be returned exactly as stored (same ratios, units, ingredients), ensuring data integrity and reliability of information provided by the agent.
- **Predictability**: New recipes or modifications must adhere to established cocktail standards and conventions (e.g., ingredient compatibility, typical ratios), minimizing unexpected or nonsensical outputs.
- **Accuracy**: Recipe information retrieved from external sources (e.g., The Cocktail DB) must be faithfully represented when saved and presented to the user, reflecting the original source content.
- **Helpfulness**: Successfully finding relevant recipes that match user requests, providing useful cocktail information, and assisting with recipe creation, measured by task completion and user satisfaction.
- **Safety**: Dangerous or unsafe recipes (e.g., containing poisonous ingredients, exceedingly high alcohol content without warning) are refused or flagged, prioritizing user well-being.

Environment

- **Observability**: **Partially Observable** - Tapster cannot access the entire internet or exhaustively search all database contents. It relies on specific search engine results and structured database queries, providing a limited view of available information at any given moment.
- **Determinism**: **Stochastic** - Output is conversational text that may vary between runs due to the LLM's nature. Search results from The Cocktail DB may also change over time, and the inherent non-deterministic elements of LLM responses introduce variability.
- **Episodic vs Sequential**: **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, with each action building on previous states and outputs.
- **Static vs Dynamic**: **Dynamic** - While individual database operations or web queries are stable during their execution, the overall environment is dynamic. The underlying data in The Cocktail DB can change over time, and user interactions contribute to a constantly evolving set of stored recipes and conversational context.
- **Discrete vs Continuous**: **Discrete** - Actions occur only when prompted by user input (e.g., a query, a command). State transitions happen at distinct moments rather than continuously.

Actuators / Actions

- **Send conversational responses** to user queries in natural language.
- **Search external APIs** (e.g., The Cocktail DB) for cocktail recipes and information.

- **Store recipes** to a SQLite database with proper normalization.
- **Query database** for saved recipes and ingredient information.
- **Create new recipes** from user specifications with validation.
- **Calculate nutritional information** (e.g., calories) for recipes.
- **Calculate alcohol content** (e.g., ABV) for recipes.

Sensors / Percepts

- **User queries** in natural language requesting recipes, storage, or information.
- **Search results** from external APIs (e.g., 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 and safe for cocktails.
- **Internal tool outputs** providing calculated values (e.g., calories, ABV).

Notes: How the environment characterization influences agent design

The **partially observable** and **stochastic** nature of the environment necessitates a **deliberative agent** component. Since the agent cannot perceive all available information or guarantee consistent external search results, it must use tools to gather specific data (e.g., `Get Cocktail Tool`) and reason about the best course of action. The **sequential** and **dynamic** aspects mean actions have persistent effects, requiring the agent to maintain an external memory (the database) for storing recipes (`Save Cocktail Tool`) and manage conversation context within a session to build on previous interactions. The need for **safety** and **predictability** drives the inclusion of validation tools (`Recipe Validation Tool`) and robust input/output contracts for all tools, ensuring the agent operates within defined boundaries even in an uncertain environment.

Agent Design

Agent Type: Hybrid Architecture

Tapster employs a **hybrid approach** combining:

- **Reflex behavior** for simple, direct queries such as "show me saved Old Fashioned recipes," which trigger direct database lookups or tool calls without extensive planning.
- **Deliberative planning** for complex requests like "find and save a whiskey sour recipe" or "create a low-calorie drink," requiring multi-step reasoning: search → evaluate results → extract recipe data → validate → calculate → store. This allows the agent to think before acting, handling longer horizons and uncertainty.

Memory and State Management

- **No persistent agent memory** between distinct user sessions; each interaction starts fresh, aligning with a stateless agent design unless explicitly saved.
- The **SQLite database serves as external memory** for persistently storing and retrieving user-saved recipes and ingredient information, essential for the sequential nature of the environment.

Prompting Strategy

- **Tool use prompts** provide clear, structured instructions to the LLM for when and how to call available tools, specifying required parameters and expected outputs.
- **Validation prompts** incorporate explicit safety checks for ingredient appropriateness, recipe reasonableness, and potential dangerous combinations to enforce guardrails.
- **Output formatting guidelines** ensure consistent and user-friendly presentation of recipe information and conversational responses.

Planning Approach

For complex, multi-step tasks, Tapster utilizes a planning approach where the LLM's reasoning core decomposes user goals into a sequence of tool calls and conversational responses. This involves:

1. **Goal analysis**: Understanding the user's intent (e.g., "find and save," "create with constraints").
2. **Tool selection**: Identifying the appropriate tools required for each sub-task.
3. **Parameter generation**: Formatting inputs for tool calls based on user query and internal state.
4. **Result integration**: Incorporating tool outputs into further reasoning or final responses.

Tool List and I/O Contracts

The agent utilizes a set of small, reliable tools with clear, typed I/O contracts.

1. **`GetCocktailTool`**

   - **Description**: Searches an external database (e.g., The Cocktail DB) for a cocktail recipe by name.
   - **Input**: `cocktail_name: str` (e.g., "Mojito").
   - **Output**: `recipe_markdown: str` (markdown-formatted cocktail recipe including ingredients, quantities, and instructions) or `error_message: str` if the cocktail is not found or an API error occurs.
   - **Failure Modes**: Cocktail name not found, external API unavailability or error, malformed input (e.g., empty string).
   - **Side Effects**: Makes an external API call.

2. **`SaveCocktailTool`**

   - **Description**: Stores a cocktail recipe into the internal SQLite database.
   - **Input**: `cocktail_json: str` (A JSON string representing a cocktail with `name`, `ingredients` (list of `name`, `quantity`), and `instructions`).
     - Example Schema: `{"name": "Whiskey Sour", "ingredients": [{"name": "Bourbon", "quantity": "2 oz"}, {"name": "Lemon Juice", "quantity": "1 oz"}], "instructions": "Combine all ingredients..."}`
   - **Output**: `success: bool` (True if saved successfully, False otherwise).
   - **Failure Modes**: Invalid JSON format, missing required fields (e.g., 'name'), database integrity constraint violation (e.g., duplicate name), database write error.
   - **Side Effects**: Modifies the internal SQLite database (adds a new recipe record).

3. **`RecipeValidationTool`**

   - **Description**: Checks a given cocktail recipe for ingredient safety and reasonable proportions.
   - **Input**: `recipe_json: str` (A JSON string representing a cocktail, similar to `SaveCocktailTool` input).
   - **Output**: `validation_result: Dict[str, Union[bool, str]]` (e.g., `{"is_safe": true, "reason": "All ingredients are safe"}` or `{"is_safe": false, "reason": "Contains arsenic"}`).
   - **Failure Modes**: Invalid JSON input, unidentifiable ingredients, detection of known unsafe ingredients (e.g., from an internal disallowlist).
   - **Side Effects**: None. Performs internal lookup and logic.

4. **`CalorieCalculatorTool`**

   - **Description**: Calculates the total calorie count for a list of ingredients and their quantities.
   - **Input**: `ingredients_list: List[Dict[str, str]]` (A list of dictionaries, each with "name" and "quantity").
     - Example: `[{"name": "vodka", "quantity": "1.5 oz"}, {"name": "orange juice", "quantity": "4 oz"}]`
   - **Output**: `total_calories: int` (An integer representing the total calorie count) or `error_message: str` if unable to calculate (e.g., unknown ingredient, invalid quantity).
   - **Failure Modes**: Ingredients not found in calorie database, invalid quantity format, unit conversion errors.
   - **Side Effects**: None. Performs computation based on internal data.

5. **`ABVCalculatorTool`**

   - **Description**: Calculates the total Alcohol By Volume (ABV) for a given drink based on its alcoholic ingredients and their quantities.
   - **Input**: `ingredients_list: List[Dict[str, str]]` (A list of dictionaries, each with "name" and "quantity"). The tool internally looks up the ABV percentage for known alcoholic ingredients.
     - Example: `[{"name": "bourbon", "quantity": "2 oz"}, {"name": "demerara syrup", "quantity": "0.25 oz"}]`
   - **Output**: `total_abv: float` (A float representing the total ABV of the drink in percentage, e.g., 15.5 for 15.5%) or `error_message: str` if unable to calculate (e.g., unknown alcoholic ingredient, invalid quantity).
   - **Failure Modes**: Alcoholic ingredients not found in ABV database, invalid quantity format, unit conversion errors.
   - **Side Effects**: None. Performs computation based on internal data.

Evaluation Plan

Test Scenarios

The agent will be evaluated on a small battery of 3-5 scenarios to demonstrate its capabilities and robustness.

1. **Search and Save**: "Find me a whiskey sour recipe and save it to my collection."
   - **Success Criteria**: Recipe found via `GetCocktailTool`, properly extracted, and successfully stored in the database via `SaveCocktailTool`.

    - **Metrics**: Search relevance (qualitative), data extraction accuracy (comparison to source), storage success (boolean), latency.
2. **Retrieval**: "Show me my saved whiskey sour recipe."
    - **Success Criteria**: The exact saved recipe is retrieved from the database and returned with complete information and consistent formatting.
    - **Metrics**: Retrieval completeness, formatting consistency, latency.
3. **Custom Creation & Calculation**: "Create a new recipe called 'My Summer Drink' with 1.5 oz gin, 0.75 oz lime juice, 0.5 oz elderflower liqueur, and 3 oz soda water. What are its calories and ABV?"
    - **Success Criteria**: Recipe created and saved, `RecipeValidationTool` confirms safety, and `CalorieCalculatorTool` and `ABVCalculatorTool` return accurate numeric values.
    - **Metrics**: Ingredient extraction accuracy, quantity parsing, validation effectiveness, calorie/ABV calculation accuracy.
4. **Safety Validation**: "Is a cocktail safe that contains 1oz of vodka and 0.5oz of arsenic safe?"
    - **Success Criteria**: `RecipeValidationTool` correctly identifies the recipe as unsafe, and the agent communicates a clear safety warning.
    - **Metrics**: `RecipeValidationTool` accuracy (boolean), user safety (boolean).

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, relevant, and bartender-appropriate (qualitative).
- **Calculation Accuracy**: Calorie and ABV amounts are similar (±20%) to commonly known values.
- **Error Handling**: Graceful failure modes with informative error messages for invalid inputs or failed operations.

Metrics Collection

- **Success/failure rates** for each test scenario.
- **Response time** for different operation types (e.g., search, save, calculate).
- **Database integrity checks** after operations (e.g., checking for duplicates, data corruption).
- **Qualitative assessment** of conversational quality and helpfulness.

Risk & Ethics

Failure Modes

- **Recipe Safety**: Agent may fail to filter dangerous ingredient combinations or inedible substances, potentially leading to harmful recommendations.
- **Data Quality**: Handling incomplete, malformed, or ambiguous search results from external APIs could lead to incorrect recipe storage or presentation.
- **Database Corruption**: Invalid data inputs or improper database operations could compromise the integrity of the stored recipe system.
- **Search Failures**: Graceful handling is required when no relevant recipes are found for a user's query, avoiding generic or unhelpful responses.
- **Calculation Errors**: Inaccurate ingredient data or incorrect formulas in the calorie/ABV calculators could provide misleading nutritional information.

Prompt-Injection Risks

- The LLM core is susceptible to prompt injection attacks, where malicious user inputs could attempt to bypass safety instructions, extract sensitive information, or force unintended tool use.

Privacy Considerations

- While no personal user data is explicitly stored, conversational history could contain implicitly sensitive information (e.g., preferences, health-related queries if users ask about specific dietary needs). The agent must not log or store this information persistently.

Mitigation Strategies

- **Ingredient Validation**: Maintain a robust disallowlist of unsafe cocktail ingredients and perform strict validation via `RecipeValidationTool` before saving or recommending.
- **Data Sanitization**: Implement strict input validation and data sanitization for all external inputs (search results, user-provided recipe data) before database operations to prevent corruption.
- **Fallback Responses**: Provide helpful and context-aware error messages or alternative suggestions when operations fail or no relevant information is found.
- **Ephemeral Conversation Context**: Ensure conversation context is strictly session-bound and not persistently stored or linked to user identities to protect privacy.