System Design: The Event Planning Agent

Version: 3.2 **Date:** October 3, 2025 **Author:** System Architect and Design Assistant

1. Introduction & Core Role

This document provides the complete technical design for the **Event Planning Agent**, a logistical execution specialist within a larger AI-driven event management ecosystem.

The agent's core role is to **convert high-level client requirements into a detailed, actionable, and data-driven event plan (the "Blueprint")**. It functions as a "second brain" for a human event planner, taking a validated client brief and performing the complex, multi-variable optimization required to find the best possible combination of vendors within a set budget. It acts as the central hub for vendor logistics and timeline creation, receiving its initial directive from a `Main AI Agent` and handing off its completed work to specialized agents for task management and client relations.

2. High-Level Architecture: Multi-Agent System

The agent is designed as a **collaborative multi-agent system**, managed by a central **Orchestrator Agent**. This architecture was explicitly chosen over a simpler linear pipeline because event planning is not a linear problem. It involves trade-offs and iterative refinement (e.g., "If we choose a more expensive venue, we must select a less expensive caterer"). A multi-agent system mimics a team of human experts collaborating to find a balanced solution.

The system is composed of four specialized agents that communicate and iterate to find optimal solutions, ensuring a separation of concerns that makes the system more robust, scalable, and maintainable.

3. Multi-Agent System In-Depth

This section details the individual agents, their tools, and the collaborative workflow they use to find the optimal event combinations.

3.1 Agent Definitions and Tools

Agent	Core Role	Key Tools & Algorithms
Orchestrator Agent	Manages the entire workflow, state, and communication between agents.	`Beam Search (k=3)` Algorithm, PostgreSQL DB for State Management

Budgeting Agent	Handles all cost allocation, financial constraint setting, and scoring.	`Gemma-2B` LLM with rule-based prompts, `calculateFitnessScore()` function
Sourcing Agent	Finds, qualifies, and ranks vendors based on various constraints.	`TinyLLaMA` LLM (Parser), Parameterized SQL Queries, Weighted Linear Scoring Model
Timeline Agent	Assesses logistical feasibility and later generates the detailed event schedule.	`Gemma-2B` LLM, Deterministic `ConflictDetection()` Algorithm
Blueprint Agent	Compiles the final, polished document after client approval.	`Gemma-2B` LLM (for prose generation and assembly)

- **Orchestrator Agent:**
    - **Role:** Acts as the "project manager" or the brain of the operation. It is responsible for initializing the process, running the main iterative loop, persisting the state to the database, and presenting the final options to the client.
    - **Tools:**
        - **Beam Search Algorithm:** This is a search algorithm that explores a graph by expanding the most promising nodes. In our context, a "node" is a partially complete event plan. Instead of exploring all possibilities (which would be computationally explosive), it keeps a "beam" of the top `k` (in our case, 3) best options at each step. This allows it to efficiently find high-quality solutions without an exhaustive search.
        - **State Management:** Interfaces directly with the `event_plans` PostgreSQL table to save and retrieve the state of the planning process.
- **Budgeting and Financials Agent:**
    - **Role:** The financial controller.
    - **Tools:**
        - **`Gemma-2B` LLM:** Used for the initial, nuanced budget allocation. It receives a prompt like: `"Total budget is $50,000 for a 300-person wedding. Allocate this budget across venue (max 12%), catering, decor, and photography. Provide 3 allocation options: 'Balanced', 'Decor-Heavy', and 'Venue-Heavy'."`
        - **`calculateFitnessScore()` Function (Detailed):** A deterministic Python function that evaluates the overall quality of a complete event combination. Its goal is to produce a single score that allows the Orchestrator to compare different combinations.
            - **Signature:** `def calculate_fitness(combination: dict, client_prefs: dict, total_budget: float) -> float:`

- **Process:** The function calculates several sub-scores and combines them using tunable weights.

Cost Score: Measures how efficiently the combination uses the client's budget. It heavily penalizes going over budget.

```
total_cost = sum(vendor['price'] for vendor in combination['vendors'])
if total_cost > total_budget:
    cost_score = 0.0
else:
    # Rewards getting closer to the full budget
    cost_score = (total_cost / total_budget) ** 2
```

-

Preference Score: Measures how well the vendors match the client's "soft" preferences (e.g., style, ratings).

```
# Pseudocode
pref_scores = []
for vendor in combination['vendors']:
    # Compare vendor attributes to client_prefs['soft_preferences']
    # e.g., style_match = 1 if client_style in vendor_style else 0
    # e.g., rating_score = vendor_rating / 5.0
    # Append a calculated score for this vendor
preference_score = sum(pref_scores) / len(pref_scores)
```

-
- **Variety Score:** A small penalty applied if the combination is too similar to others already in the Orchestrator's beam, encouraging diverse options.
- **Final Calculation:** `final_score = (0.5 * cost_score) + (0.5 * preference_score)` The weights are the most critical tuning parameters for the system's performance.

- **Sourcing and Qualification Agent:**
    - **Role:** The procurement specialist.
    - **Tools:**
        - **`TinyLLaMA` LLM:** Used as a high-speed parser. It's purpose-built for converting a string like `"I want a modern venue near Jaipur for about 300 people"` into a structured JSON: `{"hard_filters": {"location_city": "Jaipur", "capacity_min": 300}, "soft_preferences": {"style": "modern"}}`.
        - **SQL Interface:** Uses a library like SQLAlchemy to safely construct and execute queries, preventing SQL injection.
        - **Weighted Linear Scoring Model (Detailed):** This deterministic model ranks vendors *within a single category* against a specific budget allocation and client preferences.
            - **Context:** It receives a list of candidate vendors (e.g., 20 venues) that have already passed the hard filters.

            - **Process:** For each vendor, it calculates a score based on normalized features.

Price Score: How well the vendor fits the specific budget for its category.
```
# e.g., category_budget = 12000
if vendor['price'] > category_budget:
   price_score = 0.0
else:
   price_score = 1.0 - (vendor['price'] / category_budget)
```

-
- **Rating Score:** The vendor's normalized public rating. `rating_score = vendor['rating'] / 5.0`
- **Preference Match Score:** How well the vendor's attributes (e.g., style) match the client's soft preferences.
- **Final Calculation:** `ranking_score = (0.6 * price_score) + (0.2 * rating_score) + (0.2 * preference_match_score)`. The list of vendors is then sorted by this score to find the best matches.

- **Sub-Event & Timeline Agent:**
    - **Role:** The logistics and operations expert.
    - **Tools:**
        - **`ConflictDetection()` Algorithm:** A deterministic function used during the search phase. It takes a set of vendors and checks for hard constraints, returning `True` or `False`. Checks include: `venue.allows_outside_caterer == selected_caterer.is_outside`.
        - **`Gemma-2B` LLM:** Used post-selection to generate a rich, detailed draft of the master timeline, imbuing it with domain knowledge about event flow and cultural norms.
- **Blueprint Generation Agent:**
    - **Role:** The final rapporteur, invoked only after client selection.
    - **Tools:**
        - **`Gemma-2B` LLM:** Used to generate connecting prose, summaries, and format the final Event Blueprint. It receives the final structured data and a prompt like: `"You are a professional event planner. Write a comprehensive event blueprint based on the following data..."`

3.2 Collaborative Workflow

The agents work together in a loop orchestrated by the Orchestrator Agent to find the optimal solutions.

1. **Initiation:** The **Orchestrator** receives the client's request and asks the **Budgeting Agent** to create an initial set of budget allocations.
2. **Sourcing Request:** The **Orchestrator** takes one of these budget allocations (e.g., "venue budget: $12,000") and requests that the **Sourcing Agent** find a list of vendors

that meet this financial constraint, along with other client preferences.
3. **Combination Building:** The **Orchestrator** receives the list of potential vendors and starts assembling them into complete "event combinations."
4. **Validation & Scoring:** For each potential combination, the **Orchestrator** asks:
   ○ The **Timeline Agent** to perform a quick logistical feasibility check.
   ○ The **Budgeting Agent** to calculate a final "fitness score."
5. **Iteration:** The **Orchestrator**, using its Beam Search algorithm, compares the newly scored combination to its current top 3. It keeps the best ones and discards the rest. It then initiates another loop, perhaps with a slightly different budget allocation from the **Budgeting Agent**, to explore more possibilities.
6. **Conclusion:** This loop continues until a stable set of three high-quality, distinct combinations is found. The **Orchestrator** then presents these to the client.

4. Core Workflow: Iterative Combination Finding

The system's primary goal is to generate the **Top 3 optimal event combinations** for the client. This is achieved through the iterative search process detailed above.

Phase 1: Initialization

- **Step 1: Receive Initial Requirements:** The Orchestrator receives the client's form data, including the crucial **total event budget**.
- **Step 2: Initial Budget Allocation:** The Orchestrator passes the total budget and event details to the **Budgeting Agent**.

Phase 2: Iterative Search & Refinement

- **Algorithm: Beam Search (k=3):** The Orchestrator runs the collaborative loop described in section 3.2.

Phase 3: Client Selection & Finalization

- **Step 3: Present Top 3 Options:** The Orchestrator presents the three best-found combinations to the client. Each option will be a summary card including: Total Estimated Cost, Venue Name, Caterer Name, and a "Style Vibe" (e.g., "Modern & Elegant").
- **Step 4: Receive Client Selection:** The client chooses their preferred combination.
- **Step 5: Final Blueprint Generation:** The chosen combination is passed to the **Blueprint Generation Agent** to create the final document.

5. Handoff

Upon successful generation of the Blueprint, the Orchestrator initiates the handoff to downstream agents (`Task Management Agent`, `CRM Agent`).

- **To Task Management Agent:** A JSON object is sent, e.g., `{"tasks": [{"title": "Confirm booking with Venue X", "due_date": "2025-10-10"}, ...]}`.
- **To CRM Agent:** A JSON object is sent, e.g., `{"contacts": [{"name": "Venue X", "contact_person": "...", "action": "Initiate contract"},`

`...]}`.

6. Database Schema

The schema is foundational. The `event_plans` table is crucial for maintaining the state of the complex, potentially long-running planning process.

Table 1: `vendors`

```
CREATE TABLE vendors (
    -- Unique identifier for each vendor
    vendor_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    -- Official name of the vendor
    name VARCHAR(255) NOT NULL,
    -- Category of service, e.g., 'venue', 'caterer'
    service_type VARCHAR(100) NOT NULL,
    -- Cleaned city name for efficient filtering
    location_city VARCHAR(100) NOT NULL,
    -- Pre-calculated max seating capacity for venues
    max_seating_capacity INTEGER,
    -- Pre-calculated minimum veg price for venues/caterers
    min_veg_price INTEGER,
    -- Flexible JSONB column for all other vendor-specific attributes
    attributes JSONB,
    -- Timestamps for data management
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Table 2: `event_plans`

```
CREATE TABLE event_plans (
    -- Unique identifier for this specific planning job
    plan_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    -- Foreign key to a potential 'clients' table
    client_id VARCHAR(255),
    -- The current stage of the planning process for this job
    status VARCHAR(50) NOT NULL, -- e.g., 'finding_combinations', 'pending_client_selection',
'generating_blueprint', 'completed'
    -- A JSONB blob to store the state, including the top 3 combinations during search
    plan_data JSONB,
    -- The final output document after generation is complete
    final_blueprint TEXT,
    -- Timestamps for job tracking
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

7. API Specification

The agent will expose the following key endpoints for the `Main AI Agent` to interact with.

- **`POST /v1/plans`**
  - **Action:** Initiates a new planning process.
  - **Request Body:** `{"client_id": "...", "total_budget": 50000, "preferences": {...}}`
  - **Response Body:** `{"plan_id": "...", "status": "finding_combinations"}` (This is an asynchronous process).
- **`GET /v1/plans/{plan_id}`**
  - **Action:** Retrieves the current state of a plan.
  - **Response Body (while searching):** `{"plan_id": "...", "status": "finding_combinations"}`
  - **Response Body (pending selection):** `{"plan_id": "...", "status": "pending_client_selection", "options": [...]}`
- **`POST /v1/plans/{plan_id}/select-combination`**
  - **Action:** Allows the client to select their preferred combination.
  - **Request Body:** `{"selected_combination_id": "..."}`
  - **Response Body:** `{"plan_id": "...", "status": "generating_blueprint"}`

8. Implementation Sequence for New Engineer

1. **Database First:** Stand up the PostgreSQL database and create the `vendors` and `event_plans` tables using the schemas above.
2. **Build the Sourcing Agent:** This is the most foundational specialized agent. Build the logic for parsing, SQL filtering, and ranking. It can be tested independently.
3. **Build the Budgeting Agent:** Develop the logic for budget allocation and the `calculateFitnessScore` function.
4. **Develop the Orchestrator:** Implement the main application shell and the Beam Search logic. Wire it up to the Sourcing and Budgeting agents.
5. **Build the Timeline & Blueprint Agents:** Once the core search loop is working, build the final generative agents.
6. **API Layer:** Expose the functionality through the defined RESTful API endpoints.