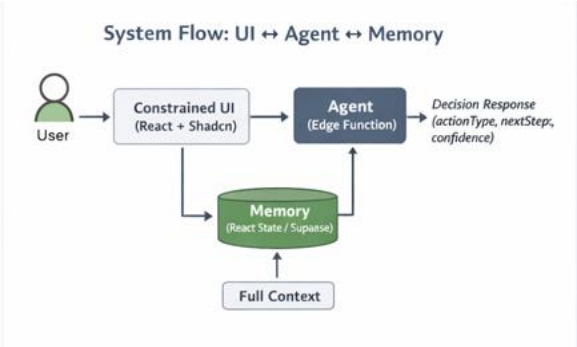
System Diagrams & Agentic Architecture

1. High-Level System Flow (UI ↔ Agent ↔ Memory)



A left-to-right flow diagram showing:

- User interacting with a Constrained UI (React + shadcn)
- The UI sending a structured JSON context to an Agent (Edge Function)
- The Agent returning a decision response (actionType, nextStep, confidence)
- The UI persisting state in Memory (React State / Supabase)
- Memory feeding the full context back to the UI for the next interaction

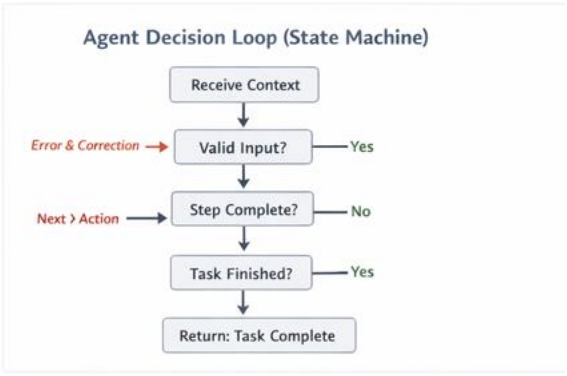
No direct arrow exists between the Agent and Memory, emphasizing that the agent is stateless.

Explanation

This diagram illustrates a strict separation of concerns. The UI is responsible for enforcing interaction constraints and rendering state, the agent is responsible only for decision-making, and memory is owned by the client. The agent never mutates UI or stores hidden state; instead, it consumes a full context snapshot and returns an explicit decision. This design prevents state drift, supports retries, and ensures deterministic behavior across sessions.

---

2. Agent Decision Loop (State Machine)



A top-down decision flow showing:

- Agent receives current context
- Validates input
- Checks whether the current step is complete
- Advances to the next step or marks the task as completed
- Returns a structured response defining the next allowed action

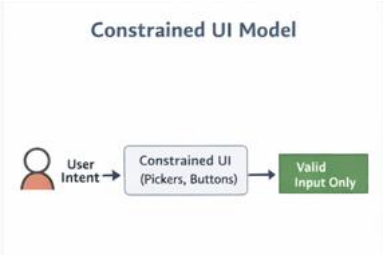
Decision points are shown as diamonds; transitions are explicit and finite.

Explanation

This diagram represents the agent as a deterministic state machine rather than a conversational chatbot. Each request results in a single, bounded decision based on the provided state. The agent does not “think ahead” or hallucinate steps; it only evaluates the current state and returns the next legal transition. This makes the system testable, auditable, and safe for complex workflows.

---

3. UI Constraint Model (Why This Is Not Plain Text Chat)



A diagram showing:

- User intent flowing into UI controls (date pickers, buttons, selectors)
- Only valid, structured inputs reaching the state model
- State passed to the agent
- Agent responses returning allowed actions back to the UI

Invalid free-text input paths are intentionally absent.

Explanation

This diagram explains how the UI prevents ambiguity before the agent is ever invoked. By replacing free-text input with constrained components, the system eliminates unclear inputs such as vague dates or incomplete answers. This dramatically reduces hallucination risk and removes the need for brittle backend parsing. The agent operates only on clean, structured data, allowing it to behave predictably.

---

4. State Model & Agent Contract

A class-style diagram showing:

- AgentContext containing taskId, currentStep, history, userInputs, progress
- AgentResponse containing actionType, nextStep, message, confidence
- A one-way dependency from context to response

Explanation

This diagram defines the strict contract between the system and the agent. The agent does not infer missing state or store memory internally. Every decision is derived from an explicit context object, and every response is structured and machine-readable. This contract enables retries, debugging, replay, and future model swaps without breaking system logic.

---

5. Failure & Recovery Flow (Resilience by Design)

A flow diagram showing:

- UI calling the agent
- Agent returning a 429 or network error
- UI presenting an error state
- Retry via user action or automatic backoff
- Same context resent to the agent

State remains unchanged during failure.

Explanation

This diagram demonstrates how the system handles real-world failures without corrupting progress. Because the agent is stateless and the client owns memory, retries are safe and idempotent. The system can recover from rate limits or timeouts without resetting the task or confusing the user, which is critical for production-grade agentic systems.

Summary (Why This Is Agentic, Not Chatbot)

This system follows modern agentic design principles: explicit context passing (MCP), deterministic state machines, constrained human input, tool-like agents, and recoverable failures. Implementing this as plain text chat would break state tracking, validation, and recovery guarantees, resulting in ambiguity, hallucinations, and brittle logic.