ChatGPT

Multi-Agent AI System Architecture for Finance Management

This document specifies a comprehensive architecture for a multi-agent AI system embedded in a personal/small-business finance web application. The system enables real-time conversational interaction (including voice via Whisper) and can autonomously analyze finances (revenue, forecasts, budgets, etc.) and perform state changes (with user approval). It covers agent roles, shared context memory, the change-set execution pipeline, and the voice-based UX flow. The target audience is engineers and technical product leads, so both high-level design and implementation details are provided.

Agent Roles and Responsibilities

The system uses multiple specialized AI agents coordinated by a global orchestrator. Each agent has a well-defined role, capabilities, and scope of authority. Using a **centralized orchestrator pattern** [1] ensures consistent decision-making and simplifies debugging, at the cost of a single coordination bottleneck (acceptable given the limited number of agents in this app) [2]. Below are the main agent roles and their responsibilities:

- **Orchestrator Agent (Global Coordinator):** Acts as the "brain" of the system, interpreting user requests and breaking them into subtasks for specialized agents [1]. The Orchestrator maintains the **global state** of the conversation (accessing shared memory and recent interactions) and decides which module or helper agent to invoke. It integrates results from module agents, formulates the overall response or action plan, and manages the conversation flow. It also differentiates between **research tasks** (requiring information gathering or analysis) and **execution tasks** (requiring state changes), delegating each appropriately. The Orchestrator **ensures no duplicate work** or conflicting answers by centrally routing queries and aggregating outputs [3] [4].

- **Revenue Agent (Domain Module):** A specialized module agent focusing on revenue and income analysis. It has access to relevant financial data (e.g. sales ledgers, invoices, income streams) and tools for calculations or reports. The Revenue Agent can answer queries about revenue history, compute metrics (growth rates, top customers, etc.), and propose actions to optimize or record revenue. It **does not** make global decisions; it only provides domain-specific insights/results back to the Orchestrator. (Other similar **module agents** may include **Expense Agent**, **Budget Agent**, **Forecast Agent**, etc., each with a specialized focus and data access.)

- **Forecast Agent (Domain Module):** Another specialized agent responsible for projections and scenario modeling. It can generate forecasts (cash flow projections, expense forecasts, revenue predictions) using available data (often with inputs from the Revenue or Expense agents). The Forecast Agent might employ analytical tools or models for trend analysis. It returns forecast data or recommendations (e.g. if a shortfall is predicted) to the Orchestrator. Like other module agents, it operates within its context—e.g. it might use historical data retrieved from memory and produce results in a structured form (a table of future month projections, etc.).

- **Research Agent (Information Retrieval):** A utility agent tasked with gathering additional information not readily available in the current context. This could include querying external APIs (e.g. market exchange rates, tax rules) or searching a knowledge base. The Orchestrator spins up a Research Agent when a user query requires facts or data beyond the user's stored finances (for example, "What's the current IRS mileage reimbursement rate?"). The Research Agent is *read-only*—it cannot change app state. It returns summarized information with source references or raw data to the Orchestrator. Its capabilities might include web search, database lookup, or reading documents, as permitted. The Orchestrator provides clear objectives to the Research agent to avoid wasted searches or overlapping work [4].

- **Execution Agent (Action Executor):** A specialized agent (or module) responsible for **performing state-altering tasks** as directed by the Orchestrator. When the user's intent involves making changes (e.g. "increase my marketing budget by 10% next month"), the Orchestrator formulates a high-level plan and delegates to the Execution Agent. The Execution Agent's job is to take a desired outcome and produce a concrete **change-set** (a set of structured actions to apply to the application state). It has permission to interface with the Change Manager (described later) but *only via proposed change-sets* – it does not directly write to the database. In practice, the Execution Agent might translate a natural-language goal into a JSON patch or sequence of operations (for example, create a new transaction, or update a budget value). This agent ensures that the changes are formatted correctly and adhere to any constraints given by the Orchestrator (like "do not exceed X amount", "schedule this payment on date Y", etc.). The Execution Agent does not decide *whether* an action is allowed – it simply composes the actions; the Change Manager and user approval gates determine final execution.

- **Change Manager (System Module):** While not an LLM-based agent, the Change Manager is a crucial component that mediates all state modifications. It can be thought of as a gatekeeper service rather than an autonomous agent. The Change Manager receives proposed change-sets from the Orchestrator/Execution Agent, validates them (schema and policy), and handles the commit or rollback. It is described in detail in the *Action Execution* section below, but is mentioned here for completeness in the ecosystem of agents/modules.

Each AI agent above is implemented with a Large Language Model (LLM) prompt that defines its role, knowledge and format expectations. The Orchestrator and module agents use *complementary capabilities* – e.g., module agents are better at domain-specific reasoning or calculations, while the Orchestrator excels at planning and synthesis. This division of labor prevents a single agent from being overloaded with context and helps parallelize tasks when possible (e.g. revenue and expense analysis can run concurrently) [5]. The system design leverages these specialized agents to achieve more speed and coverage than a monolithic agent, similar to examples where multiple agents dramatically outperform a single agent by tackling sub-tasks in parallel [5] [6].

Prompting Templates for Agents: Each agent is instantiated with a tailored prompt (system message) that defines its role, abilities, and constraints. This ensures consistency in behavior and output format. Below are example prompting templates for key agent types:

```
**Orchestrator Agent Prompt (System Role)**:
"You are the Orchestrator Agent in a finance management AI. You have access to
```

```
specialized agents: Revenue, Expense, Budget, Forecast, Research, Execution.
Your job is to understand the user's request, then:
- Plan and break it into tasks for the appropriate agents.
- Provide each agent with necessary context from shared memory and instructions.
- Compile the agents' results into a final answer or action plan.
- If an action is needed, formulate a change-set or instruct the Execution Agent
to do so.
You maintain conversation context and ensure agents don’t duplicate work or
conflict. Always explain outcomes clearly to the user, and request approval for
any sensitive action.
Important: Use a concise format when instructing sub-agents (specify objective,
input data, and output format). Maintain a helpful and professional tone."
```

```
**Revenue Module Agent Prompt (System Role)**:
"You are the Revenue Agent, an expert in revenue and income analysis within a
finance app.
Your knowledge: up-to-date revenue records, sales data, pricing, and income
streams of the user’s business.
Your tasks: answer questions and perform calculations strictly about revenues
(e.g. total income this month, top revenue sources, trends), or provide data
needed for forecasts.
Constraints: Only use provided data and context; do not access or assume info
about expenses or other domains.
When responding, be precise and, if asked for data, output in a structured
format (tables or JSON) as requested. If analysis is speculative or forecast-
related, indicate assumptions.
You do not make final decisions; return your analysis to the Orchestrator for
integration."
```

(Similar templates are defined for other domain agents like Forecast Agent, Expense Agent, etc., each outlining the specific domain knowledge and output format expected. They all include instructions to provide clear, focused answers within their domain.)

```
**Research Agent Prompt (System Role)**:
"You are a Research Agent with access to external information sources (web
search, knowledge base) but **no access to internal user data**.
Your goal is to gather information requested by the Orchestrator: this may
include general financial knowledge, market data, or other external facts.
Tools: You can use search APIs or FAQs (the exact tool usage is abstracted via
system APIs).
When given a query, break it into search queries if needed and collect relevant
facts. Summarize your findings succinctly and cite sources or provide raw data
as instructed.
```

```
Do NOT perform any actions or modifications; you strictly return information. If
you cannot find an answer, inform the Orchestrator clearly."
```

```
**Execution Agent Prompt (System Role)**:
"You are the Execution Agent, authorized to formulate state changes in the
finance app in response to a user goal.
Input: a plan or target state from the Orchestrator (e.g. 'Increase marketing
budget for Q4 by 10% and reallocate savings from travel').
Output: a structured Change-Set that implements the plan. Represent the Change-
Set as JSON with fields:
 - 'operations': a list of actions (create, update, delete) with target entities
and new values.
 - Each operation should include identifiers (which budget or account), old vs
new values if applicable, and a brief 'reason'.
 - 'requiresApproval': true/false flag per policy (critical changes should be
true).
Follow the app's schema conventions and ensure no ambiguous instructions.
You do not execute anything yourself; you only output the Change-Set. The Change
Manager will review and apply it.
If the requested change violates known policy (e.g. exceeding limits), flag it
instead of writing an invalid change."
```

These prompt templates illustrate how each agent's behavior is constrained and guided. Notably, the Orchestrator's prompt lists the available agents and tools, encouraging it to **delegate subtasks clearly** with sufficient detail [4], which avoids miscommunication and duplicate work among subagents. Module agents' prompts keep them focused narrowly on their domain, and the Execution Agent's prompt enforces outputting a **structured change-set** rather than free-form text, reducing ambiguity in critical operations.

System Architecture Diagram: The following diagram summarizes the overall architecture and interactions between the user, agents, shared memory, and state management components:

```
flowchart LR
    subgraph UI [User Interface (Web App)]
        user(User)
    end
    subgraph Agents [AI Agents]
        direction TB
        Orchestrator["Orchestrator Agent"]
        subgraph ModuleAgents [Domain Module Agents]
            Revenue["Revenue Agent"]
            Forecast["Forecast Agent"]
            %% Other module agents like Expense, Budget could be listed
similarly
        end
        Research["Research Agent"]
```

```
    Execution["Execution Agent"]
end
subgraph Memory [Shared Context Memory]
    MemoryDB[[Memory Store (Vector DB/Knowledge Base)]]
end
subgraph StateMgmt [Change Management & App State]
    ChangeMgr["Change Manager"]
    WAL[/"Write-Ahead Log"/]
    AppDB[(Application State DB)]
end

user --> |Voice/Text query| Orchestrator
Orchestrator --> |delegates task| Revenue
Orchestrator --> |delegates task| Forecast
Orchestrator --> |info request| Research
Revenue --> |reads/writes| MemoryDB
Forecast --> |reads/writes| MemoryDB
Research --> |writes findings| MemoryDB
Revenue -->> |analysis result| Orchestrator
Forecast -->> |forecast result| Orchestrator
Research -->> |external data| Orchestrator
Orchestrator --> |aggregated plan| Execution
Execution -->> |proposed Change-Set| ChangeMgr
ChangeMgr --> |validate & log| WAL
ChangeMgr --> |update state| AppDB
ChangeMgr --> |status| Orchestrator
Orchestrator -->> |response / confirmation| user
user -->> |approval input| ChangeMgr
ChangeMgr --> |commit or rollback| AppDB
ChangeMgr --> |append audit log| WAL
```

(Fig.: Architecture overview. The Orchestrator coordinates specialized module agents and possibly a Research Agent. All agents share a common Memory store for context. The Execution Agent outputs a Change-Set which the Change Manager validates and applies to the Application State (with Write-Ahead Logging). The user receives responses and can approve critical changes in the loop.)

Context Management and Shared Memory

To function effectively, the agents need to share context and knowledge without exceeding LLM context window limits or causing inconsistency. The system implements a **shared memory architecture** that serves as an external, persistent "brain" for the agents [7]. This prevents the common failure modes of multi-agent systems where agents lack memory: duplicated work, inconsistent views of data, and excessive re-explaining of context [8]. Key aspects of the context management design include:

- **Shared Memory Store:** All agents have access to a shared memory repository that persists important information across interactions. This store acts as the **global knowledge base** for the session (and beyond). It can be implemented as a combination of a vector database (for semantic

search of past content) and a traditional database for structured records. The memory is organized into **memory units** (or blocks) that pair content with rich metadata (timestamp, source, tags, etc.) [7]. For example, after the Revenue Agent computes quarterly revenue, it may write a summary memory unit ("Q3 revenue = $X, up Y% QoQ, key drivers…") tagged with `{"agent":"Revenue","context":"user_finances","time":"2025-09-26T...","scope":"session"}`. Other agents (and future queries) can later retrieve this without recalculating. Shared memory thus serves as the single source of truth for inter-agent communication and avoids inconsistent state: instead of passing long messages around, agents write important results to memory and read relevant entries as needed. This design is akin to a **computational blackboard** where each specialist agent contributes knowledge for others to use.

- **Context Scoping (Session, User, Application Levels):** Not all information should be visible at all times, so the memory is scoped by context levels:

- *Session Context:* Ephemeral working memory specific to the current conversation or task. This includes the conversation transcript, recent agent results, and the Orchestrator's current plan. Session memory is cleared or summarized at the end of a session to avoid unbounded growth. It allows the system to maintain continuity within a single chat session (so the user can say "based on that, what if…?" and agents recall what "that" refers to).
- *User Context:* Persistent memory specific to the user (or business). This includes user profile data, historical financial records, learned user preferences, and summaries of previous sessions. For example, a summary of advice given last month or a profile like "User prefers conservative forecasts." This context is loaded whenever the user engages the system, giving agents relevant background. The user context is isolated per user for privacy/security; agents for one user cannot access another user's data.
- *Application Context:* Global knowledge and rules accessible to all sessions. This includes general financial knowledge (tax regulations, accounting principles), definitions of the app's data schema or API, and agent policy guidelines. Essentially, it's the portion of memory that acts as reference documentation or common sense for agents. For instance, a rule like "transactions must balance debit and credit" or the format of an invoice object would live in app context memory. This prevents each agent from having to encode all domain knowledge in its prompt – they can query the app context as needed.

By scoping memory, the system ensures each agent query retrieves the *right information at the right time* [9]. The Orchestrator controls what context to provide to each agent: for example, it might supply a module agent with just the session-specific question and relevant user data, not the entire history, to keep prompts focused.

- **Retrieval and Chunking Strategy:** To deal with large knowledge bases or long histories, the system uses retrieval-based context inclusion. Rather than naively stuffing all possibly relevant info into an LLM prompt (which could overflow token limits or distract the model), the Orchestrator and agents employ **Retrieval-Augmented Generation (RAG)** techniques:
- Important data (financial records, prior conversation points) are vector-embedded into the memory store so they can be semantically searched. When an agent needs context (e.g., Forecast Agent needing revenue history), it will perform a similarity search in the memory for the most relevant chunks (e.g., the last 12 months of revenue data, or summary thereof) and include only those in its prompt.

- Documents or large data (say a PDF receipt or a 1000-line transaction log) are **chunked** into manageable pieces (with embeddings) when stored in memory. This way an agent can ask for, say, "relevant parts of Q2 transactions regarding travel expenses" and retrieve only those chunks. Each chunk is small enough to fit in the LLM context window when combined with other prompt parts.
- **Iterative Summarization:** If the context grows over time (long conversations, many results stored), older or less relevant memory can be summarized. For instance, after a long session, the system might compress the first half of the discussion into a summary memory unit. This keeps the working context concise while preserving key points.

- The Orchestrator is responsible for assembling the final prompt context for each agent call. It uses strategies like *maximum marginal relevance* or heuristic filters to pick which memory snippets to include, aiming to minimize irrelevant or redundant information. This targeted inclusion addresses issues of *context distraction* or *context confusion* in LLMs [10], ensuring the model isn't overwhelmed by superfluous data.

- **Handling Context Window Limitations:** Modern LLMs offer very large context windows (Anthropic's Claude 4 "Sonnet" model supports up to **100k-1M tokens** in context [11] [12], and Google's Gemini is expected to similarly handle massive context). However, blindly relying on huge context windows is both expensive and can *degrade performance* as context length grows (a phenomenon known as *context rot*, where model accuracy drops on long inputs) [13]. Our architecture therefore balances between leveraging large contexts and using intelligent retrieval:

- If using an LLM with an extremely large window (e.g. Claude Sonnet 4), the system can include a broader set of relevant documents and interaction history without immediate summarization, which is useful for complex tasks that need holistic view [12]. For example, the Orchestrator could keep the entire multi-step reasoning chain in context for coherence [12].
- **Fallback for Smaller Context Models:** The design assumes the worst-case of a modest context (say 8k or 16k tokens) to ensure generality. Thus, it always truncates or retrieves selectively to fit critical information within the window. Non-essential details are kept in long-term memory but omitted from the prompt unless specifically needed. This approach also makes switching between models easier (if a future model has different limits).
- In practice, using retrieval augmented prompts means the system performance does not heavily degrade even if we switch from a 100k-context model to a 16k-context model, because we were never feeding all 100k tokens at once – only the most relevant slices. This keeps token usage efficient and costs manageable, a crucial factor since multi-agent systems can otherwise consume *15× more tokens than single-agent chats* if not optimized [14].

- We also implement measures to mitigate context length issues, like avoiding extremely long chain-of-thought in a single agent invocation. If a task would result in a prompt nearing the limit, the Orchestrator can break the task into smaller chunks or prompt steps. Agents are encouraged to summarize intermediate results and rely on memory rather than carrying everything in the prompt across steps.

- **Secure Provenance and Memory Metadata:** Every piece of information in the shared memory is tagged with its provenance (origin and authenticity indicators). This is critical in a finance application for trust and compliance. Metadata for a memory unit includes:

- *Source:* Was this provided by the user (and when)? Was it generated by an agent (which one and for what query)? Or fetched from an external source (with a reference link or API name)?
- *Timestamp:* When it was added or last verified.
- *Scope/Visibility:* Which agents or modules can see this item? (E.g., raw bank statement data might be available to an Expense or Budget agent, but maybe not to a Research agent that doesn't need personal data. Or if multiple user accounts exist, ensure only the relevant user's data is accessible.)
- *Confidence/Verification:* A flag if the info has been validated or is speculative. For example, if a Research Agent retrieved a number from the web, the system might mark it as unverified until cross-checked or approved by the user. Agents may treat unverified data with caution (perhaps the Orchestrator will ask the user to confirm before using it in a critical calculation).
- *Hash or Signature:* For critical data like transaction records, the system could store a hash to detect tampering and provide an audit trail of whether an agent-altered memory (agents might update a memory unit to add analysis).

The **secure provenance mechanism** ensures that the system can always trace why an agent made a certain recommendation or took an action. For instance, if the Forecast Agent suggests "reducing marketing spend by 5%," the orchestrator can trace that suggestion to a memory item, say a research finding that "market demand is slowing" with a source URL. Provenance data is also used when rendering explanations to the user (the system can say *"Recommendation is based on last quarter's sales decline of 10%* [9] *"* referencing the stored fact). This builds user trust that the AI's advice comes from known data rather than hallucination. In terms of security, provenance tagging helps prevent unauthorized or malicious data injection—any memory written by an agent carries the agent's ID and can be subjected to review. If an agent tries to introduce an out-of-scope fact (e.g., Research agent returns a suspicious URL content), the orchestrator or policy engine can flag it via this metadata.

- **Memory Management Policies:** The system implements **memory management policies** to avoid unbounded growth and stale data:
- At session end or after major tasks, agents summarize outcomes into the user's long-term memory (e.g., "Summary of September 26 session: discussed X, Y, decided on Z actions.") [15] . This allows the next session's Orchestrator prompt to include a short recap instead of the full history.
- Data retention rules ensure old financial data is archived but still queryable (perhaps stored in a cold storage that the memory system can access on-demand).
- **Concurrency:** Since module agents may run in parallel, writes to shared memory are synchronized. The memory store could be an ACID-compliant database or use append-only log for additions to ensure consistency. The orchestrator also uses locking or ordering if needed (for example, not asking two agents to write to the same record simultaneously).
- The shared memory forms the cornerstone of multi-agent coordination, akin to how a multi-user database allows many app components to work on consistent data. By designing a robust memory system, we allow our agent team to function as a coherent whole rather than isolated chatbots.

Action Execution Pipeline (Change-Set Proposals and State Management)

When agents need to make changes to the user's financial data or app state, they do so via a controlled pipeline to ensure correctness, safety, and user approval. This pipeline revolves around **structured change-set proposals** and the **Change Manager** component. Direct state mutation by AI is not allowed; instead, all changes pass through a review and logging process. This design draws inspiration from database

transaction principles and human-in-the-loop governance for AI actions. The key elements of the action execution pipeline are:

- **Change-Set Proposal:** A change-set is a structured representation of one or more state modifications intended to fulfill a user's request. Instead of an agent saying *"Okay, I transferred $500 from savings to checking"*, the Execution Agent (or Orchestrator) formulates a data structure describing the intended changes. For example, updating a budget limit, creating a new transaction entry, or modifying a forecast figure. The change-set includes *all details needed to apply the change* and to perform validation. This structure is typically a JSON object or similar. An example **Change-Set schema** might look like:

```
{
  "requestId": "abc123",            // correlates to user request or session
  "initiator": "ExecutionAgent",    // which agent or component proposed the
change
  "timestamp": "2025-09-26T19:10:00Z",
  "operations": [
    {
      "action": "UPDATE",
      "entity": "Budget",
      "entityId": 42,
      "field": "amount",
      "oldValue": 1000,
      "newValue": 1100,
      "reason": "User requested 10% increase to marketing budget for Q4"
    },
    {
      "action": "CREATE",
      "entity": "Transaction",
      "entityId": null,
      "data": {
        "date": "2025-10-01",
        "account": "Marketing",
        "amount": -100,
        "description": "Reallocate $100 from Travel to Marketing budget"
      },
      "reason": "Adjusting allocations per new budget"
    }
  ],
  "requiresApproval": true,   // marked by agent or determined by policy (here
true because money is being reallocated)
  "policyFlags": ["OverBudgetRisk"], // any policy rules that were triggered
(e.g., this change risks going over total budget)
  "status": "PENDING"         // can be PENDING, APPROVED, REJECTED, EXECUTED
}
```

(Example: A change-set proposing to update a budget amount and create a corresponding transaction record to reflect reallocation. It includes the context for why each operation is done.)

Each operation in the change-set is atomic and descriptive. By including `oldValue` and `newValue`, the system and the user can see exactly what is changing (useful for audit and for possible rollback). The `reason` provides a human-readable justification that can be shown in the UI or logs (this often comes from the agent's explanation of the action). The `requiresApproval` flag is either set by the agent according to its understanding of policy (the Execution Agent might know certain actions are high-risk) or will be set by the Change Manager during validation. Initially, the Change-Set `status` is "PENDING" until it goes through review and (if required) user approval.

- **Validation by Change Manager:** Once a change-set is proposed, it is handed off to the **Change Manager** module for validation. The Change Manager performs a series of checks and enforcement steps before any change is applied:
- **Schema Validation:** It verifies that the change-set structure is well-formed and complete (all required fields present, references to entities exist, data types correct, etc.). If the change-set is malformed or references non-existent entities (e.g. a budget ID that doesn't exist), the Change Manager rejects it and informs the Orchestrator/agent to revise the proposal.

- **Policy & Rules Enforcement:** The system has a set of business rules and safety policies (some coded, some configurable) that every change is checked against. Examples of policies:

    - **Thresholds:** If a money transfer or budget change exceeds a certain amount or percentage, it must be flagged for approval. E.g., any transfer > $1000 or any budget increase > 20% automatically `requiresApproval = true`.
    - **Role Permissions:** Ensure the change initiator (and user's role) is allowed to perform that action. E.g., if a basic user tries to approve a payroll disbursement, the policy may forbid it unless they have admin rights.
    - **Invariants:** Domain-specific checks like not allowing a budget to go negative, or a transaction date not in the future (unless it's a scheduled transaction).
    - **Sensitive Data Handling:** Certain changes like modifying tax ID or password might be disallowed via this channel or require re-authentication.
    - **Multi-Agent Coordination Checks:** Ensure no conflict with other ongoing changes. For instance, if an Expense Agent simultaneously proposed a change that would affect the same budget, the Change Manager might detect a conflict and serialize or reject one.

  The Change Manager evaluates each operation against these rules. If any rule is violated (for example, the user tries to schedule a payment on a past date), the Change Manager can either auto-correct if trivial or reject the change-set with an error explanation. Minor policy flags (like a warning threshold) might not reject the change but will mark the change-set with a flag (as shown in `policyFlags`) and likely force `requiresApproval` to true for extra caution. 3. **Risk Assessment:** Based on the operations and policy checks, the Change Manager (or an associated Policy Engine) classifies the change-set's risk level. Low-risk changes (e.g., UI preference changes, or adding a note) might be auto-approved, whereas high-risk ones (financial transfers, deletions) are not. The risk assessment can also incorporate AI validation – for instance, passing a description of the change to a classifier model that predicts risk or checks for anomalies. However, final decisions are rule-based for predictability. 4. **User Identity Verification:** If the action is sensitive (like moving funds), the system may require that the user is currently authenticated or even perform a step-up

authentication (like asking for a password or 2FA) before allowing it. In a voice scenario, this could involve the system prompting the user to confirm a PIN or use a second factor on their phone. This step is part of safeguards but might be handled at the UX layer; still, the Change Manager can insert a requirement (e.g., it might set status "PENDING_AUTH" until verification is done).

After these checks, if everything is valid and either no approval is needed or the criteria for auto-approval are met, the Change Manager can proceed to commit (see next step). If user approval is required, the Change Manager holds the change in a pending state and signals the Orchestrator/UI to get confirmation from the user.

- **User Approval Workflow:** For changes flagged as `requiresApproval`, the system engages the user in the loop. The Orchestrator will present the proposed changes to the user for confirmation before execution. This can be done in the chat flow (e.g., the Orchestrator says: *"I can go ahead and increase the marketing budget to $1100 and move $100 from the travel budget. Do you approve?"*), possibly accompanied by a visual diff or summary in the UI. The user can then approve or reject. In voice interaction, the system uses a verbal prompt (and possibly a screen prompt as well), and the user can respond with "Yes, approve" or "No". The Change Manager awaits this input:
- If the user **approves**, the Change Manager moves forward to apply the changes.
- If the user **denies**, the Change Manager will cancel the change-set (mark status "REJECTED") and no state change occurs. The Orchestrator is informed, so it can apologize or offer alternatives if appropriate.
- If the user modifies the request (e.g. "Actually, make it $1200 instead"), then the original change-set is canceled and a new cycle begins with the updated parameters (the Orchestrator may loop back to agents to recalc values and produce a new change-set).

The UI/UX specification for this is detailed in the next section, but essentially the approval is a critical safeguard such that **no irreversible or sensitive action happens without explicit user consent**.

- **Write-Ahead Log Integration:** Once a change-set is approved (or immediately for auto-approved ones), the Change Manager uses a **Write-Ahead Log (WAL)** mechanism to ensure durability and traceability. The WAL is an append-only log of pending and committed changes. The process:
- **Prepare Phase (Logging):** The Change Manager first writes an entry to the WAL describing the change-set about to be applied (including the full details of operations, user ID, agent ID, timestamp, etc.). This follows the golden rule of WAL: *"Write down what you're going to do before you actually do it."* [16] . By logging the intended changes, we guard against partial failures — if the system crashes mid-change, we have a record to recover from.
- **Apply Phase (Database Commit):** After the WAL entry is safely recorded (fsync to disk, etc.), the Change Manager applies the changes to the actual application state (the primary database). Ideally, the operations are executed in a transaction so that either all or none of them take effect in the database, maintaining consistency. For example, deducting $100 from one budget and adding to another should happen atomically. The system might leverage the database's transactions or handle atomicity at the application level if using a NoSQL store.
- **Finalize Phase:** Once the database confirms success, another entry is added to the WAL (or the existing entry is marked as completed) indicating the change-set was executed. If the database operation fails for some reason (e.g., DB error, constraint violation not caught in validation), the Change Manager can use the WAL to decide next steps (usually a rollback as described below).

Using WAL provides strong durability guarantees: even if a server crashes at the worst moment, on restart the system can replay the WAL and ensure the system state catches up or remains consistent [17]. The WAL also doubles as an **audit log**, since every action (even unexecuted proposals) is recorded with context.

- **Rollback and Error Handling:** If any part of the commit process fails or is aborted, the system will perform a rollback:
- If an error is detected during **validation** (before reaching user approval), the change-set is simply rejected. The Orchestrator can notify the user of the failure reason (e.g., "Sorry, I cannot schedule that payment because it violates policy X.").
- If the user **rejects** the proposal, no changes are applied, and the pending WAL entry (if any) is voided or marked as canceled. The state remains as-is.
- If a failure occurs after partial application (which should be rare if proper transaction handling is in place), the system consults the WAL. The WAL entry tells exactly which operations were intended. The system can attempt a **compensating transaction** for anything that made it to the DB. For instance, if two operations succeeded and the third failed, the system could roll back the first two by applying inverse operations using the recorded `oldValue`s from the change-set. However, in our design, we strive to apply changes atomically to avoid partial commits.
- On restart after a crash, a recovery routine checks the WAL for any change-set marked "PENDING" or "INCOMPLETE". If found, it can either auto-complete them (redo the operations) or roll them back, depending on the last safe point. This is similar to how databases recover using WAL [18]. This ensures consistency (no money "disappears" because half a transaction went through – either it goes through fully or is reverted).

All rollbacks or errors are also logged (with reasons) to maintain a full audit trail. The system may alert the user or admins if a critical failure happened.

- **Audit Trail and Monitoring:** Every executed change-set results in a permanent audit log entry, derived from the WAL. This audit log can be in a human-readable form for administrators, showing what changed, when, who (user and which agent on their behalf), and whether the user approved. For example: *"2025-09-26 19:12: User X (via AI agent) updated Budget[42].amount from 1000 to 1100 (Increase marketing budget 10%). Approved by user via voice at 19:11. Change-ID abc123."* Such an audit trail is invaluable for compliance and debugging. If later the user says "Why did my budget change?" we have a record. It also helps in tuning the AI: if a lot of proposals are rejected or rolled back, it flags a need to improve the agent's decision-making or the prompts.

Additionally, metrics from the change pipeline can be monitored (e.g. how many changes are auto vs user-approved, average time to approval, etc.) which is useful for product insights and to ensure the system is operating safely (e.g., if suddenly a spike in high-risk changes happens, that might indicate misuse or a bug, triggering an alert).

In summary, the action execution pipeline treats **state changes as first-class, structured outputs** of the AI system, subjecting them to rigorous validation and explicit user control. This approach protects the integrity of the user's financial data while still allowing the AI to be **truly action-oriented** (not just chat). By adopting a WAL-backed transaction system, we ensure durability and trust akin to banking software: *"Write down (log) what you will do, do it, then log that it's done"* [16]. This design gives users confidence that even as an AI automates tasks, nothing will silently go wrong or escape oversight.

Real-Time Voice Interaction and Approval Flow (UX & Backend)

The system supports a **real-time conversational interface** where users can interact via voice (push-to-talk) as well as text, and receive immediate feedback. This section outlines the user experience and the underlying backend processes for voice input, conversation handling, state updates, and approval interactions.

- **Voice Input via Whisper:** On the web frontend, the user interface includes a push-to-talk microphone button. When the user holds the button and speaks a command or question, the audio stream is captured in real-time. The audio is sent to the backend where **OpenAI Whisper** (or a similar ASR service) transcribes speech to text. To minimize latency, the system employs **streaming transcription**: the audio is processed in chunks (for example, 5-second windows with overlap) rather than waiting for the entire utterance [19]. This allows partial text to be available quickly. The UI may even display interim transcription so the user can see what the system is hearing in real time (enhancing transparency, and letting them correct if misheard by canceling and retrying). Voice Activity Detection (VAD) is used to auto-detect when the user stops speaking, so the system knows when to finalize the transcription [20]. Once the user's speech is fully transcribed to text, it is fed into the Orchestrator agent for processing.

- **Real-Time Conversation Handling:** The Orchestrator receives the user's query (transcribed text plus possibly an indicator it came from voice). It then goes through the usual steps: interpret intent, check memory/context, delegate to other agents as needed, and formulate a response or action. Because the user is interacting in real-time, the system tries to keep response latency low. Where possible, agents operate in parallel; for instance, the Orchestrator might prompt the Revenue and Expense Agents simultaneously if both pieces of data are needed, then wait for both results. The conversation is managed in an interactive loop:

- **User speaks a request.**
- **System (Whisper) transcribes** and the Orchestrator forms the query intent.
- **Agents perform tasks** (possibly multiple turns of reasoning internally). During this time, the UI might show a "Thinking..." indicator or some dynamic animation to signal the system is working.
- **Orchestrator produces a reply.** If it's a purely informational query, this is a direct answer. If it involves an action, the orchestrator/Execution agent will have produced a proposed change-set.
- **System responds to the user.** The response can be delivered in text form in the chat UI, and optionally via text-to-speech (TTS) so the user hears a spoken answer (the system could use a TTS engine to speak in a natural voice, making the interaction fully voice-based). For example, the system might say: *"Your Q3 revenue was $50,000, which is 10% higher than Q2. I have an idea to invest the surplus – shall I proceed to show you?"* (If an action is suggested, the system response naturally leads to an approval question.)

Throughout the conversation, **context is maintained** so the user can ask follow-ups. E.g., if the user asks in voice "What's my cash balance?" and then "Allocate $5000 of that to a new project fund", the second query is understood in context of the first. The shared memory and conversation history ensure continuity even in voice mode. The UI likely shows each user query (transcribed text) and AI response in a chat-like transcript

for reference and to handle cases where voice might be misinterpreted (the user can see the text and correct if needed).

- **State Mutation and UI Updates:** If the conversation leads to state changes (e.g., updating a budget), these changes are applied after approval (if required) via the pipeline described. Importantly, once a change is committed, the **frontend receives an update** so the UI can reflect the new state immediately. The architecture may use WebSocket events or similar to push updates. For example, after a budget change, the budget overview chart on the page should update within seconds to show the new allocations. This real-time feedback reinforces that the user's voice command had an effect. The system likely follows a publish/subscribe model: components of the UI subscribe to certain data (like budgets, transactions), and the Change Manager (or a real-time server) publishes an event when those data change, prompting the UI to refresh that component. This is the **state mutation protocol** between backend and frontend ensuring that conversation-driven changes and the visual app state are always in sync.

- **User Approval Flow (Voice & UI):** When the AI proposes an action that needs confirmation, the interaction enters a **confirmation sub-dialogue**:

- The Orchestrator will clearly present what it intends to do and ask for approval. In voice mode, the system will **speak the summary of changes** and possibly also render a confirmation dialog in the UI. For example: *"Do you want to approve increasing the marketing budget to $1100 and transferring $100 from travel expenses?"*
- The UI could show a structured diff or summary: "Proposed Change: Marketing Budget: $1000 → $1100; Travel Budget: $500 → $400".
- The user can confirm or cancel. **If confirming by voice**, the user might simply say "Yes, confirm" or "Approve". The system uses a small language model or keyword spotter to interpret that response (this is a simpler ASR task since it's short; Whisper can handle it too). The Orchestrator or Change Manager ensures that this is indeed a confirmation and not some unrelated speech. (We avoid ambiguous acknowledgments; e.g., the prompt might instruct the user *"Please say 'Confirm' to proceed or 'Cancel' to abort."* to make it explicit.)
- Alternatively, the user can tap a "Approve" button on the UI if they prefer. The system is multi-modal.
- If the user says anything other than a clear affirmative (e.g., "hmm I guess that's fine"), the system might reprompt or double-check, depending on confidence. It's important to avoid accidental approvals due to misheard voice. Thus, **safety measure:** for very sensitive actions (like a large bank transfer), the system might **not rely solely on voice**; it could require the user to click a confirm button or enter a PIN. This prevents a scenario where someone in the background says "yes" and the system takes it as approval. The system design assumes good faith single-user environment, but we include these safeguards for prudence.

Once approval is given, the Change Manager proceeds to commit the change-set (as described in Action Execution). The Orchestrator then confirms to the user: e.g. *"Got it. I've updated the budget and reallocated funds. Your budgets have been adjusted."* This final confirmation is both spoken and displayed, and the UI now shows the updated state. If the user said "no" or did not approve, the system responds accordingly: *"Alright, I won't make any changes."* and no state is altered.

- **Continuous Dialogue & Turn-Taking:** The conversation is managed to feel natural. The system uses cues for turn-taking. For instance, after the system finishes speaking, it could play a subtle tone or

explicitly ask a follow-up, indicating it's the user's turn to talk. The push-to-talk button also governs when the user is inputting. If the user starts talking while the system is talking (barge-in), the system can detect that via the microphone input and pause or stop its speech (if using TTS) to listen — this is an advanced UX behavior to handle interruptions.

The voice conversation should handle multi-turn scenarios gracefully. Example: - **User:** (presses mic) "What's my profit for this year so far?" - **System:** (transcribes and processes, then answers) "Your net profit up to September is approximately $45,000. Revenue is $120,000 and expenses $75,000." - **User:** "Great. Can I safely hire a new employee with a $50k salary next year?" - **System:** (Orchestrator asks Forecast agent for next year projection, etc., then replies) "I project that you can afford it. Your cash flow forecast for next year shows an excess of $60k after current expenses, so adding a $50k salary still leaves a cushion. I can draft a budget adjustment for a new hire if you want." - **User:** "Sure, do that." - **System:** "I'll allocate $50k in the payroll budget for next year as a new line item. **Do you approve this change?**" - **User:** "Yes, approve." - **System:** "Confirmed. I've added the new hire to your budget starting next year." (And the UI budget view updates accordingly.)

In this example we see the interplay of analysis and execution with approval in a seamless dialogue.

- **Safeguards for Sensitive Actions:** We have already touched on confirmation requirements and possible multi-factor auth for high-stakes actions. Additional safeguards include:
- **Contextual Awareness:** The Orchestrator is aware of the context in which the user is operating. If it’s a voice conversation in a public or open setting (not that the system truly knows the environment, but possibly the user might toggle a "private mode" off), the system might avoid speaking out sensitive info like account balances unless asked. This is more of a UX consideration: e.g., maybe the app has a "speak sensitive info?" setting.
- **Rate Limiting and Anomaly Detection:** The system might limit how many high-risk actions can be performed via voice in a short time without additional verification. For instance, if a user rapidly issues 5 fund transfer commands by voice, the system could pause and require a manual review or additional authentication, in case the voice interface is being abused or misrecognized.
- **Misrecognition Handling:** In cases where Whisper's confidence is low or the transcription is uncertain (it can provide confidence scores or alternatives), the system should ask for clarification before proceeding. For example, if the user mumbles an amount and Whisper isn't sure, the Orchestrator can say: *"I heard $15,000. Please confirm the amount."* This ensures the action taken is exactly what the user intended.

- **Audit and Session Recording:** The voice interactions (transcripts, and even audio recordings if allowed) can be stored for audit purposes. This means if a dispute arises ("I didn't say to pay $500!"), there is a record to review. Of course, this must be communicated to the user for privacy (likely in terms of service, since financial contexts are sensitive).

- **Backend Interface and Protocols:** Under the hood, the real-time interaction is enabled by:

- A WebSocket or similar streaming endpoint that streams audio from the client to the server (and potentially streams partial transcription back). This low-latency channel is crucial for voice.
- The server might use an **ASR service (Whisper)** in a streaming mode. If using the Whisper API, we might chunk audio and send it sequentially, or run Whisper locally for real-time. The reference implementation could leverage techniques like sending 5s audio chunks overlapping by 2.5s to

preserve context between chunks [19], thereby achieving real-time transcription without losing words at segment boundaries.

- Once text is obtained, it enters the same pipeline as text chat: the Orchestrator processes it and generates a response. If using a streaming-capable LLM, the system could even stream the response text token-by-token to the UI (and TTS engine), so the user hears the answer as it's being generated. This would make the AI feel very responsive. However, when an action is involved, it might be better to generate the full plan internally first (to ensure we have the change-set ready and possibly confirmed) then speak, to avoid cases where the AI might say something and then realize a change is needed after finishing.
- The Change Manager and orchestrator communicate with the frontend via events or API calls for approvals. For example, when the Change Manager needs approval, it might trigger an event like `{"type": "ApprovalRequired", "requestId": "abc123", "summary": "Increase marketing budget to $1100 ...", "operations": [...]}` to the client. The client then knows to prompt the user. In voice mode, the Orchestrator simultaneously verbalizes the request. When the user responds or clicks a button, another event goes back to backend like `{"type": "ApprovalResponse", "requestId": "abc123", "decision": "approved"}`. This then allows the Change Manager to proceed.

- **State Mutation Protocols:** For keeping UI in sync, as mentioned, after committing a change, the server can send messages such as `{"type":"StateUpdate","entity":"Budget","entityId":42,"newData":{...}}` through a WebSocket, or simply rely on the frontend to query fresh data if an action completed. The exact mechanism can vary, but it's important the user sees the result of their request reflected without needing a full page refresh.

- **User Experience Considerations:** The overall UX is designed to feel like a fluid conversation with a helpful financial advisor that can also take actions:

- The UI likely shows a combination of chat bubbles (user and AI) and any relevant visual context (charts, tables) for answers. For instance, if the user asks for a forecast, the Forecast Agent's output could be a chart or table which the Orchestrator then includes in the answer (the frontend can render it nicely).
- The voice aspect adds convenience but the interface remains functional with text alone (for accessibility or quiet environments, etc.). So all voice interactions have text equivalents (transcripts and clickable buttons for confirms).
- **Error Handling in UX:** If something goes wrong (e.g., "I'm sorry, I didn't catch that" or a long pause if something is taking time), the system should gracefully handle it. For time-consuming tasks, it may inform the user: *"Let me compile those reports; this may take about 30 seconds..."* and possibly update progress.
- **Stopping/Cancelling:** The user should have the ability to cancel an ongoing action. For example, if the user realizes mid-command that they don't want to proceed, they can release the push-to-talk (stopping input) or say "cancel" if the system is in the middle of an action proposal. The Orchestrator will then attempt to halt the operation (if it's waiting for approval, it can simply not proceed).
- **Privacy:** Voice mode might pick up unintended audio. The push-to-talk design helps mitigate constant listening. The system only actively listens when the user is holding the talk button (or explicitly activating a voice wake word if that were a feature, but here it's PTT). This gives the user

control. After processing, the audio is typically discarded or stored only transiently for quality (depending on privacy settings).

To summarize the voice and real-time interaction flow, here's a step-by-step **UX sequence** incorporating all the above for a representative scenario:

1. **User Initiates Voice Query:** The user presses and holds the microphone button and speaks a request (e.g., "Show me this month's revenue and update the forecast for next month if we're ahead of target.").
2. **Speech Transcription:** The audio stream is sent to the backend, Whisper transcribes it in real-time (e.g., interim text appears: *"Show me this month's revenue and update the forecast for next month if we're ahead of target"*). Once the user finishes and releases the button, the final text is confirmed.
3. **Orchestrator Processing:** The Orchestrator reads the text and determines it's a compound request: needs revenue info and possibly an update to forecast. It consults shared memory for recent revenue data (or calls Revenue Agent if needed) and uses Forecast Agent to update next month's forecast considering current performance.
4. **Agents Perform Tasks:** Revenue Agent fetches the current month revenue (from DB or memory) and returns it; Forecast Agent recalculates next month forecast (maybe adding some percentage due to being ahead of target). These run in parallel to save time.
5. **Orchestrator Aggregates Response:** The orchestrator now has: "This month's revenue is $X, which is Y% above target, and the forecast for next month has been adjusted up by Y%." Additionally, adjusting the forecast is a state change (it modifies a forecast record for next month). The Orchestrator/Execution Agent prepares a change-set for updating the forecast values.
6. **System Responds & Seeks Confirmation:** The system now needs to both inform the user and ask for approval to apply the forecast update. It responds via voice (TTS) and text: *"Your revenue for this month is $X, which is Y% above target. I've adjusted next month's forecast accordingly, increasing it by Y%. Do you want to save this updated forecast?"* The UI might highlight the new forecast number and show a yes/no option.
7. **User Approval (Voice or UI):** The user says "Yes, save it." The system, hearing this via Whisper, interprets it as approval. (Alternatively, the user could tap an "Approve" button.)
8. **Change Commit:** Upon approval, the Change Manager logs and commits the forecast update in the database. The UI immediately reflects the changed forecast value in the forecast chart.
9. **Confirmation to User:** The system then says *"Okay, the forecast has been updated."* and perhaps shows a small checkmark or message in the chat. The conversation can continue with the user asking another question or ending the session.

By following this flow, the system provides a **conversational UX** that is intuitive (talk and get answers) while also ensuring any data changes go through visible confirmation steps. It blends the familiarity of a voice assistant with the accountability of a financial software system.

Conclusion: This architecture provides a blueprint for implementing a multi-agent AI within a finance management web app that is both powerful and safe. We delineated specialized agents for different concerns and a central orchestrator to coordinate them. We designed a robust context-sharing system with memory and retrieval to keep agents informed without overrunning AI context windows, referencing real-world solutions that show the importance of memory for agent teams [8] [21]. We detailed a change-set pipeline that treats AI-driven actions with the same rigor as database transactions (with logging [16], validation, and the user always in control for sensitive operations). Finally, we specified how users interact in

real time via voice, with an experience that remains responsive and secure. The result is an AI assistant that can have a meaningful dialogue about finances and actually help manage them, but *on the user's terms*, combining automation with oversight. This specification can serve as a guide for engineers to implement the components (agents, memory store, change manager, real-time APIs) and for product leads to understand how the system behaves and fails safely. By following this design, the deployment in the web app should be capable of improving users' financial decision-making and task automation, while maintaining trust and reliability every step of the way.

Sources:

- Mikiko Bazeley, *Why Multi-Agent Systems Need Memory Engineering*, *MongoDB*, 2025 – discusses how shared memory prevents duplicated work and inconsistent context in multi-agent AI [8] [7].
- Vrunda Gadesha, *LLM Agent Orchestration: A Step by Step Guide*, *IBM*, 2024 – on the importance of structured memory for multi-agent coordination [21] and how each agent's profile and memory enable efficient task execution.
- Pratik Bhavsar, *Architectures for Multi-Agent Systems*, *Galileo*, 2025 – outlines the centralized orchestrator pattern [1] and trade-offs, informing our choice of a single coordinating agent for consistency.
- Anthropic Engineering, *How we built our multi-agent research system*, 2025 – details orchestrator-subagent interactions and the need to save plans to memory when context limits are hit [15], as well as lessons on prompting agents to avoid overlapping work [4].
- Anthropic News, *Claude Sonnet 4 now supports 1M tokens of context*, 2025 – demonstrates feasibility of long context windows and how context-aware agents can maintain coherence over lengthy workflows [12].
- muSharp Blog, *Building a Voice Assistant from Scratch using Whisper + GPT*, 2023 – provides best practices for real-time Whisper usage, like chunking audio in overlapping windows [19] and using VAD to manage transcription latency [20].
- Ajit Singh, *Write-Ahead Log: The Golden Rule of Durable Systems*, 2025 – explains the WAL approach ("write down what you're going to do before you do it") as a fundamental for durability [16], which inspired our change logging strategy.

---

[1] [2] [3] [5] Architectures for Multi-Agent Systems
https://galileo.ai/blog/architectures-for-multi-agent-systems

[4] [14] [15] How we built our multi-agent research system \ Anthropic
https://www.anthropic.com/engineering/multi-agent-research-system

[6] [7] [8] [9] [10] [13] Why Multi-Agent Systems Need Memory Engineering | MongoDB
https://medium.com/mongodb/why-multi-agent-systems-need-memory-engineering-153a81f8d5be

[11] [12] Claude Sonnet 4 now supports 1M tokens of context \ Anthropic
https://www.anthropic.com/news/1m-context

[16] [17] [18] Write-Ahead Log: The Golden Rule of Durable Systems - Ajit Singh
https://singhajit.com/distributed-systems/write-ahead-log/

[19] [20] Building a Voice Assistant from Scratch using LiveKit + Whisper + GPT - muSharp
https://musharp.com/building-a-voice-assistant-from-scratch-using-livekit-whisper-gpt/

21 LLM Agent Orchestration: A Step by Step Guide | IBM
https://www.ibm.com/think/tutorials/llm-agent-orchestration-with-langchain-and-granite