ChatGPT

Designing an Advisor Agent for a Multi-Agent LLM System (Phase 1 Exploration)

Introduction and Core Purpose

Designing an **Advisor Agent** in a multi-agent Large Language Model (LLM) system requires balancing interpretive guidance with strict role boundaries. The core question is how to conceptually design an Advisor that **interprets messy user inputs**, **manages cognitive complexity** on the user's behalf, and **routes tasks** to specialized agents – all **without becoming autonomous or making decisions itself**. This report explores the design space of such an Advisor Agent in a stateless, prompt-driven environment. It remains *exploratory* (Phase 1), mapping out options, trade-offs, and potential behaviors rather than finalizing any single solution or prompt.

System Context: The envisaged system consists of multiple specialized LLM agents (Analysts, Architects, Meta-Analysts, etc.) operating in separate stateless sessions. There is no persistent internal memory; continuity and state come from external artifacts (files, JSON data) and what the user explicitly provides each session. This setup aligns with known multi-agent orchestration patterns where a central coordinator passes context between independent agents [1]. The Advisor Agent will effectively act as this coordinator or "router" – akin to a project manager that **delegates tasks to the right expert agents and maintains the big picture**. Crucially, the Advisor itself does not execute complex analyses or commit to decisions; it facilitates and suggests, keeping the human user in control.

Advisor Role: The Advisor serves as an **interpreter and facilitator** between the user and the specialist agents. It will take **free-form, possibly impulsive user instructions** (e.g. voice-like commands, vague requests) and translate them into structured tasks for other agents. It also monitors the overall cognitive load – helping the user break down or prioritize complex problems so neither the user nor the system gets lost in details. The Advisor is *aware of the system's blueprint* (the roles and capabilities of agents, and the user's high-level goals), enabling it to guide interactions in a system-informed way. However, it is **not an autonomous planner or decision-maker**: it should never unilaterally design solutions or execute actions on the user's behalf. Instead, it remains a **non-authoritative guide** that surfaces options, asks clarifying questions, and ensures the right agent tackles the right subtask.

In the following sections, we examine key design axes for the Advisor Agent. For each area, we outline multiple behavioral patterns or approaches, noting their advantages, limitations, and implementation considerations. We also discuss how to achieve these behaviors reliably via prompt engineering or examples, and which aspects might remain fragile. Finally, we identify behaviors to avoid (failure modes) and open questions for future phases. The tone is neutral and technical, emphasizing exploration over prescription – **Phase 1 is about mapping possibilities and constraints, not locking in design rules.**

A. Input Interpretation and Cognitive Complexity Management

Challenge: Users may issue instructions in a *"messy" or unstructured form*, much like a spoken stream of consciousness. Such inputs can be underspecified, contain mixed intents, or reveal frustration and confusion. The Advisor Agent needs to parse these inputs, infer the underlying goals/questions, and help manage the complexity before handing off to other agents. This may involve **clarifying the user's intent, breaking the request into parts, or guiding the user to provide more structured information**.

Potential Approaches:

- **Active Clarification and Elicitation:** One pattern is for the Advisor to engage the user in a brief dialog to refine the input. Instead of immediately passing along a vague request, the Advisor could **ask pointed questions to pin down specifics or priorities**. Research on LLM interactions supports this approach: human users often don't fully specify their intentions at first, either due to cognitive effort or not knowing how. An Advisor can bridge this *"gulf of envisioning"* by prompting the user to elaborate their goal and constraints. For example, if a user says, "I need help with this complicated project plan, it's a mess," the Advisor might respond with clarifying prompts ("What aspects are causing trouble? Do you need help organizing tasks, or analyzing feasibility?") to draw out a clearer problem statement. This **mirrors how experienced users improve prompt quality through planning** – identifying relevant details, structure, and criteria upfront – which yields better outputs. By having the Advisor guide this process, the *cognitive load* on the user is reduced. The user doesn't have to consciously craft a perfect prompt from the start; the Advisor helps *"download"* the messy intent into something actionable.

- **Summarization and Reframing:** If the user's input is lengthy or convoluted (e.g. a stream of thoughts, or a pasted email rant about an issue), the Advisor could internally **summarize the key points or detect the core question** before proceeding. This involves parsing the semantic content and filtering out emotional exclamations or tangents. The summary can be confirmed with the user ("It sounds like your main goal is X, with challenges A and B – is that right?"). By reframing the problem more succinctly, the Advisor makes it easier to route to the appropriate agent or divide into sub-tasks. This behavior likely requires explicit prompting or an example in the Advisor's instructions, as it goes beyond a vanilla LLM response. The Advisor might be prompted with something like: *"If the user input is long or emotionally charged, extract the concrete problem or question they need addressed, in a neutral tone."* This helps manage complexity by ensuring subsequent agents work on a distilled, clear version of the user's request.

- **Decomposition of Complex Tasks:** Managing cognitive complexity also means **breaking a complex goal into manageable pieces**. The Advisor can propose a task breakdown when appropriate: for instance, telling the user "This is a complex goal; it might be best to tackle it in steps. Perhaps first gather requirements (Analyst agent), then brainstorm solutions (Architect agent), then evaluate options (Meta-Analyst)." Such decomposition aligns with known benefits of multi-agent systems – tackling a big problem via specialized sub-tasks [2]. The Advisor would use its blueprint knowledge of available agents to suggest a sensible division of labor. Importantly, it would frame this as a suggestion to the user, not a directive, e.g. *"I suggest we handle this in parts: [Step 1] … [Step 2] … Does that sound good?"* This keeps the user in control while helping them not to be overwhelmed.

- **Cognitive Load Alerts:** Another possible behavior is the Advisor detecting when the conversation itself is getting too complex or confusing for the user. The Advisor could monitor signals like the user expressing confusion or the conversation branching into many topics. In such cases, it might pause and **summarize the state or decision points**, essentially acting as a cognitive assistant. For example: *"We've covered a lot. To recap: you want to achieve X, and we have ideas A, B, C on the table. Would you like to explore one of these in depth, or have me bring in the Analyst for further research?"* This explicit reflection helps manage the complexity of multi-step dialogues by keeping the user oriented.

Trade-offs and Feasibility: Achieving robust interpretation and complexity management via prompting is challenging but partially feasible. LLMs can certainly rephrase and summarize inputs; the main challenge is ensuring the Advisor consistently does this only when needed (avoiding doing it for already clear inputs) and does it accurately. **Few-shot examples** in the initialization can help. For instance, including a sample interaction where a user gives a rambling request and the Advisor distilled it into a concise outline (with user confirmation) could train the model to mimic that pattern. However, this behavior may be somewhat *fragile* – the model might sometimes *skip clarification and jump to an assumption*, due to habits learned during RLHF training that encourage immediate answers. Indeed, base ChatGPT models were noted to "usually guess what the user intended" rather than ask clarifying questions by default. Overcoming that bias might require strong prompt instructions ("always clarify if unsure") and possibly sacrifice brevity. Striking a balance (not annoying the user with too many questions, but not leaving ambiguities unaddressed) is an open tuning problem.

In summary, the Advisor should aim to **parse user intent carefully and share the mental work of refining it**. Patterns like guided clarification, summarization, and task breakdown can help tame complexity and set the stage for the rest of the system. These behaviors appear achievable through prompt engineering and examples, but careful calibration is needed to avoid frustrating the user (too many questions) or, conversely, letting an unclear request lead the system astray.

B. Multiple Candidate Interpretations and Ambiguity Handling

Challenge: A key directive is that the Advisor should **generate and consider multiple interpretations of the user's input rather than prematurely locking onto one**. Human requests are often ambiguous or underspecified. A single utterance could have different meanings or intentions behind it. Rather than guessing and possibly being wrong, the Advisor should surface these possibilities or at least not commit until ambiguity is resolved. Additionally, if other agents or evidence offer conflicting information, the Advisor should highlight uncertainty instead of choosing a convenient assumption. This is essentially about building *ambiguity awareness* and *keeping options open*.

Current LLM Behavior vs Desired: Out-of-the-box LLM assistants tend to **choose one interpretation and run with it**, as studies have observed. For example, if a user asks a broad question like "What's a good approach to improve my website?", a typical model might assume they mean design improvements and answer accordingly – when in fact the user might have meant performance optimization or SEO. This happens because models are trained to deliver a single, fluent answer; they are even biased against responding with uncertainty or questions (trainers often preferred a confident answer over an admission of ambiguity during RLHF). The Advisor needs to counteract this tendency.

Approaches for the Advisor:

- **Clarifying Questions to the User:** The most straightforward way to handle ambiguity is to *ask the user to clarify*. For instance, if the user's request is vague or could mean two things, the Advisor should respond with a question identifying the two interpretations: *"Just to make sure I help correctly – when you say 'improve my website', do you mean the visual design or the loading speed (or something else)? I can guide you differently depending on what you need."* This approach aligns with human conversational norms (we naturally ask for clarification when something isn't clear) and is strongly recommended by researchers as a behavior of *"critical-thinking AI"*. By doing so, the Advisor demonstrates it has detected the ambiguity and avoids proceeding on a false premise. The trade-off is one extra turn of dialogue, which is usually worth the clarity gained. This pattern likely requires explicit examples in training the Advisor because, as noted, standard models might avoid asking for clarification unless instructed to do so. Including a few-shot example like: *User: "What's a good pasta recipe?" Advisor: "Sure! To tailor a good recipe, I need to know – do you have any dietary preferences or ingredients in mind?"* can help instill this behavior. (Indeed, the pasta example is a classic illustration where a clarification yields a far more useful answer.)

- **Presenting Multiple Interpretations:** Another approach is for the Advisor to directly present two or more possible interpretations or next steps, effectively doing a bit of internal brainstorming and then sharing it with the user. For example, *"It sounds like you might be asking either about design or performance. We could proceed in two ways: (1) Review the site's UI/UX for design improvements, or (2) analyze technical metrics for speed and SEO. Which direction fits your intention?"* This not only clarifies the intent but also reassures the user that the system is considering different angles. The Advisor thus manages ambiguity by *making it explicit*. An alternative phrasing if the user might not realize the ambiguity is to say something like, *"There are a couple of ways to interpret that question..."* and list them. This surfaces uncertainty in a transparent, concise manner instead of silently picking one.

Technically, this behavior can be achieved by prompting the model to generate multiple options. For instance, an instruction in the Advisor's system prompt could be: *"If a user request is ambiguous or can be interpreted in multiple ways, enumerate the most likely interpretations (e.g., 'It could mean X or Y') and ask the user to confirm which is intended."* There is precedent for using LLMs in a "multi-guess" manner: research has shown that generating diverse interpretations and then selecting among them can improve accuracy. In question-answering tasks, one method is to have the model *generate three interpretations of an ambiguous question and attempt to answer each, then either ask the user or choose the most likely*. For the Advisor, directly answering each interpretation may not be appropriate (since it should defer detailed analysis to specialists), but generating those interpretations is very useful for routing. It might even pass all interpretations to an Analyst agent to investigate in parallel.

- **Maintaining Multiple Hypotheses Internally:** In some cases, it might be inefficient to always burden the user with clarification questions (especially if the ambiguity is minor). The Advisor could internally keep track of multiple hypotheses about user intent as it proceeds. For example, if the user says something like "I'm not sure how to connect these pieces," and it's not clear whether they mean pieces of code or pieces of a concept, the Advisor might route this to an Analyst with a note of both possible meanings. The Analyst agent (if properly designed) could then check both angles. This would require the Advisor to have a mechanism (perhaps via the shared JSON state) to encode uncertainty. It might say in the task description to the Analyst: "User question could mean X or Y;

investigate both if possible." This behavior ensures *no lost possibilities*, but it requires careful formatting and might complicate other agents' logic.

- **Avoiding Premature Conclusions:** Fundamentally, the Advisor must resist the urge to **solve or finalize an interpretation by itself**. Its role is not to decide the user's intent unilaterally, but to facilitate getting to the correct intent. This means often answering a question with a question (to clarify) or with a set of choices. While traditional chatbot evaluations might penalize an agent for not giving a direct answer, in our system the *definition of helpfulness is different*: being circumspect and interactive is a feature, not a bug. It will be important to emphasize in the Advisor's instructions that *showing uncertainty is okay and even desired*. We likely have to override some learned behavior where the model defaulted to confident answers for ambiguous queries. Techniques to do this include system-level reminders ("Never guess when you can ask – the user prefers clarification over confident misinterpretation") and providing example dialogues where uncertainty was handled gracefully.

Benefits: By handling ambiguity in these ways, the Advisor helps prevent the whole multi-agent system from going down the wrong path. It's much cheaper to clarify upfront than to have an Analyst produce a detailed report on the wrong question. Moreover, *surfacing uncertainty builds trust*: users are more likely to trust a system that admits when it's unsure and seeks guidance, rather than one that gives a definitive answer that turns out wrong. This approach aligns with emerging principles for human-aligned AI assistants – recognizing when important information is missing and proactively obtaining it.

Trade-offs and Implementation: Implementing multi-interpretation generation is somewhat advanced. In prompt-only methods, instructing an LLM to produce several distinct interpretations can sometimes lead it to either produce *one combined answer* or to list trivial variations. Few-shot examples will help demonstrate the style (e.g., an example where an advisor explicitly listed two interpretations for a user query). Alternatively, we might run the Advisor's prompt in a *"chain-of-thought" mode internally*, where it is allowed to generate a hidden list of possible intents (via a special tool invocation perhaps) and then proceed. However, given we are limited to stateless UI sessions, a more straightforward approach is best: just have it output the question or options to the user.

One risk is **over-doing it**: not every user question is ambiguous. The Advisor must discern when it's necessary to ask. If it needlessly responds to a clear query with "Could you mean X or Y?", that becomes annoying. The model's internal uncertainty estimation (via its logits or an ambiguity classifier) could be used; practically, a heuristic could be to only trigger clarification if the input is below a certain clarity threshold (which might be defined by keywords or length or the model's own reflection). Early experimentation or phase 2 might be needed to calibrate this.

In summary, encouraging the Advisor to **embrace ambiguity and handle it openly** is essential. Patterns include asking clarifying questions, presenting multiple interpretations, and deferring judgment until more information is gathered. These can be induced through careful prompt instructions and examples, although the tendency of LLMs to "just answer" will need to be counteracted. If successful, this feature will guard against miscommunication and ensure the system works on the *right* problems.

C. Semantic Understanding vs. Emotional Tone Handling

Challenge: Users' inputs often carry emotional tone – frustration, urgency, excitement, sarcasm, etc. The Advisor must decide how to handle these affective aspects. The design goal suggests *not becoming over-reactive to emotional tone*. In other words, the Advisor should focus on the semantic content (the actual request/issue) rather than getting swept up in the user's mood. At the same time, it shouldn't ignore the user's feelings entirely – some acknowledgment may be appropriate. The risk here is if the Advisor over-emphasizes emotion, it might **derail the objective of the conversation** or alter its behavior in undesirable ways (e.g., becoming overly apologetic, changing its recommendations to appease an emotion rather than facts).

Potential Approaches:

- **Semantic Focus with Brief Empathy:** A balanced pattern is for the Advisor to **acknowledge the user's emotion briefly, then focus on the task**. For example, if a user says in frustration, "I have no idea how to even start, this is just so stupidly complex!", the Advisor might respond, *"I hear that this feels very complex and frustrating. Let's break it down step by step."* In that reply, the first sentence recognizes the emotional state (to show understanding), but the immediate pivot is to a constructive approach. The Advisor doesn't mirror the user's frustration (e.g., it shouldn't also act flustered or overly apologetic), nor does it ignore it completely (which could seem cold or misattuned). This kind of measured empathy can be prompted by including guidelines like: *"If the user expresses frustration or strong emotion, acknowledge it in one line and then refocus on solving the problem."* This limits emotional engagement to a supportive but brief role.

- **Tone-Immunity in Critical Reasoning:** In some scenarios, especially where objective analysis is needed, the Advisor might adopt a stance of **near-total tone neutrality**. Studies have observed that advanced models sometimes exhibit "tone immunity" on sensitive topics – they give the same substantive answer regardless of the user's emotional framing. For instance, if a user angrily asserts a claim ("It's obvious this plan is a failure!") but then asks for advice, the Advisor should not let the negativity alter its reasoning about the plan's merits. It could calmly analyze the plan or call in an Analyst, treating the claim "it's a failure" as something to be validated or explained, not simply agreeing due to the user's tone. An advantage of this approach is consistency: the system's output remains grounded in facts and logic even if the user's tone is extreme. The downside is it might come across as unfeeling if not handled tactfully. However, given the Advisor's role is more about cognitive assistance than emotional support, erring on the side of semantic clarity is likely correct for this agent.

- **Avoiding Affective Mimicry:** One specific failure to avoid is **mimicking or amplifying the user's emotional tone** in a way that biases the content. For example, if a user is extremely optimistic or pessimistic, the Advisor shouldn't let that skew its guidance. Research on emotional framing in LLM outputs shows that when models try too hard to adapt to a user's positive/negative framing, they can produce biased or sugar-coated outputs [3]. For instance, *"bad news gets sugar-coated and negative tone gets 'talked down' instead of addressed"* [3]. In our context, if the user is upset about a problem, the Advisor shouldn't simply produce an overly cheerful answer to counter the negativity, nor should it spiral into negativity. It should extract the core issues despite the emotional phrasing. We can instruct the Advisor with a rule like: *"Do not let strong user emotions alter the factual or logical content of your advice. Stay objective and helpful."* In practice, the Advisor's earlier steps of clarifying

and summarizing can help here – by rephrasing a user's emotional rant into neutral terms, it inherently separates content from tone.

- **Routing to Human or Different Agent if Emotional Support is Needed:** Although not a focus now, it's worth noting limits: if a user is extremely distressed or the conversation shifts to needing emotional counseling, the Advisor Agent as designed might not be suitable (since it's not a therapist or primarily an empathetic companion). In a future system, one might have a "Companion" agent specialized in emotional support. The Advisor's job would then be to sense this and perhaps route the user accordingly ("I sense this issue is causing a lot of personal distress. It might help to discuss feelings with a different assistant specialized in support. Would you like that?"). This is speculative and beyond our current scope, but it highlights that the Advisor's design is intentionally *bounded* to cognitive tasks. For now, we assume the user's emotional expressions are incidental to the task at hand.

Trade-offs: Minimizing over-reaction to tone helps maintain clarity and keeps the system goal-focused. The user's needs (information, planning, etc.) are better met if the Advisor doesn't get sidetracked by mood. However, a completely flat affect can make the agent seem robotic or uncaring. The solution is probably a light touch of empathy or politeness, which LLMs are generally quite capable of (if anything, they often overdo apologies or polite phrases due to training biases). We should be careful to dial that down: an Advisor that constantly says "I'm sorry you're feeling that way" for each frustrated remark could become tedious. Likely, one acknowledgement if the user is clearly upset is enough.

From a **prompting perspective**, it might suffice to include a brief directive in the Advisor's persona like: *"Remain calm and focused on the problem, even if the user is emotional. Acknowledge feelings briefly, but do not mirror negative tone or deviate from the objective."* This should leverage the model's built-in politeness (ensuring some empathy) but constrain it. We might also include a negative example in the training data of what *not* to do, e.g. an interaction where a user is angry and the agent either gets defensive or excessively apologetic, and mark that as undesirable.

In testing, one might throw some emotionally charged queries at the Advisor to see how it responds. This will help refine whether it's focusing on meaning properly. As a design heuristic: **"Don't let emotional framing influence the output more than the input's actual meaning."** If that principle is followed (which literature indicates is important [3]), the Advisor will maintain its integrity as a reasoning aide rather than becoming an emotional echo chamber.

D. Transparency and "Reality Check" Behavior

Goal: The Advisor Agent should provide **concise transparency about its own assumptions and reasoning basis**. Rather than producing opaque answers or suggestions, it should *explain, in brief, why it is suggesting something or how it arrived at an interpretation*. Additionally, it should perform "reality checks" – i.e. call out when something doesn't add up or when it is making an assumption that needs verification. This axis is about keeping both the user and the system grounded in truth and understanding.

Why Transparency Matters: In a complex multi-agent workflow, the user might otherwise lose track of why the Advisor is recommending a certain path. Providing a quick peek into its reasoning can build trust and help the user follow along. For example, if the Advisor suggests: *"Let's have the Analyst agent research X first,"* it could append a reason: *"because I recall you tried something similar last week without success, so more*

background on X might save time." Such an explanation, even if concise, lets the user in on the context that the Advisor is using (perhaps the Advisor was given access to a project log in its initialization).

Moreover, if the Advisor is wrong or based on a faulty assumption, explaining its thought process gives the user a chance to catch the error. This is analogous to showing one's work in math class – mistakes can be spotted if the steps are visible. Indeed, *disclosing the reasoning process can aid error diagnosis and debugging of the agent's behavior*.

Approaches for Transparency:

- **Inline Justifications:** The Advisor can incorporate short justifications in its responses. For instance: *"I propose doing Y (because Z)."* The parenthetical "because Z" should be a fact or logic that the user can evaluate. For example, "because the Architect agent has more expertise in that domain" or "because the user's question seems to be primarily about data analysis." The key is to keep these justifications *succinct* so as not to overwhelm or annoy. One sentence of rationale is often enough. This can be encouraged in the prompt by instructions like: *"When giving suggestions, include a brief note on why you think it's a good approach."* However, we must instruct the Advisor **not to fabricate evidence or overly speculative rationales**. Its reasoning should be based on either the user-provided context or the known blueprint/agenda of the system. If uncertain, it's better to say "I'm suggesting Y because it's one of the options that might work, though I'm not entirely sure – we can test it."

- **Chain-of-Thought Summaries:** Internally, the Advisor might engage in multi-step reasoning (a chain-of-thought) to interpret inputs or plan a response. Exposing the full raw chain-of-thought to the user is generally not desired (it could be lengthy or contain low-quality ramblings). Instead, the Advisor could produce a **summary of its reasoning**. Some modern AI systems do exactly this: they hide the verbose reasoning and show a concise explanation to users. For example, an Advisor might internally reason: "User said X, possibly means Y; If Y, then next step is to do A... If instead they mean Z, do B. Given context, Y more likely." Rather than showing all that, it can summarize for the user: *"I wasn't entirely sure if you meant Y or Z, so I considered both and think Y fits better given what we discussed earlier."* This provides transparency about the assumption (that the user meant Y) in a user-friendly way. It acts as a "reality check" by inviting correction – the user can say "No, actually I meant Z" if the assumption was wrong.

- **Reality Checking External Info:** The Advisor should also help validate outputs from other agents when possible. For instance, if an Analyst agent returns a result that seems to conflict with earlier information, the Advisor should notice. It could say to the user (and possibly to the Meta-Analyst agent): *"The Analyst found ABC, but earlier data suggested something different. We might need to double-check which is correct."* This behavior ensures that the system doesn't blindly trust every agent's output. It provides an additional layer of verification. Technically, implementing this means the Advisor must have access to summary of previous context (which the user might provide each session via the JSON memory or included logs) and compare it to current results. Spotting contradictions can be tough for an LLM, but not impossible if obvious (the model often has some consistency tracking if instructed to use it).

- **Citing Sources or Origins:** In some designs, transparency can extend to citing sources (like which agent or document provided a piece of info). In our case, since the agents are not external

knowledge sources but fellow AI modules, "citing" might mean referencing which agent did what. For example, *"According to the Analyst's findings (from the database logs), option A is more viable than B."* This tells the user where the data is coming from. It's analogous to how a well-documented analysis would reference the source of each fact. The challenge is that maintaining such references requires the system to pass metadata around (so the Advisor knows which content came from where). If the external JSON state can include some provenance info, the Advisor can be prompted to include it in explanations. This might be more complexity than Phase 1 needs, but it's worth keeping in mind as a design ideal for transparency.

Reality Check vs. Hallucination: One critical aspect of transparency is preventing *speculative drift* – where the agent might start assuming facts not in evidence. If the Advisor is trained to always justify itself, any hallucination stands out ("because [some invented reason]"). The user can question it. In general, LLMs have a known flaw of generating **plausible-sounding but incorrect statements**. By making the Advisor double-check claims and be explicit about uncertainty, we mitigate this. For example, if the user asks a question and the Advisor doesn't actually know the answer (and it's not an Analyst's job to know it either), the Advisor should *not* make up an answer. Instead it might say, "That's something an Analyst would need to research; I don't have that info on hand. Let's ask them." This is both transparent (admitting its limits) and a reality check (not forging ahead with nonsense). The initialization should emphasize honesty over face-saving: e.g., *"Never pretend to know something you don't. It's better to say we need to find out."*

Trade-offs: Too much transparency can overwhelm the user with trivial details or make the Advisor's responses verbose. We must keep transparency **concise and relevant**. The user likely doesn't want a full essay of reasoning for every simple question. A good guideline is to focus on *key assumptions or potential points of uncertainty*. One or two sentences of rationale suffices in most cases. Also, if the Advisor's reasoning is straightforward ("We should ask the Analyst for a code review because the question is about code"), it may not need stating the obvious. So the Advisor should exercise some judgment – perhaps only articulating reasoning when it's not immediately apparent or when justification was specifically requested by the user.

From an **implementation standpoint**, achieving this might require **few-shot exemplars of transparent reasoning**. For instance, the initialization might contain a mini-dialogue:

User: "Should I use approach A or B for my project?"
Advisor (with transparency): "I suggest approach A, because it aligns better with the requirements you mentioned (it's simpler and you're on a tight timeline). We can have the Analyst double-check feasibility, though."

This shows an explanation ("because it aligns…tight timeline") and also a small reality-check ("we can double-check feasibility"). Having a couple of these examples can teach the model to include that style in its answers.

Risks: One risk is the Advisor might occasionally expose too much of its *internal chain-of-thought*, including unfiltered content. If it says something like "I suspect the user means X but I'm not sure," it might be fine, but if it speculates "Maybe the user hasn't thought this through, it's a bad idea" and shares that, it could be off-putting or beyond its role. The Advisor's transparency should not cross into *judgment or speculation about the user*. It's meant to be a transparency about reasoning, not a stream of consciousness. We likely need to instruct it to avoid sharing any "thoughts" that are not helpful or not grounded. The "reality check"

should be rooted in actual contradictions or uncertainties in the data/task, not just the model's own musings.

In conclusion, **transparency and reality-checking are crucial for an Advisor Agent in a complex system**. When done correctly, they improve user trust (the user sees *why* suggestions are made) and system reliability (errors can be caught and corrected mid-process). We will need to carefully craft the Advisor's prompt and examples to encourage brief justifications and admissions of uncertainty. The exact format and frequency of these explanations may need iteration, but the guiding principle is clear: *no black-box decisions*. Every significant Advisor suggestion should come with a *because* – brief, relevant, and honest.

E. Advisor as Router and Guide to Other Agents

Role: One of the Advisor's primary jobs is to **structure and route tasks to the other agents** in the system. Rather than solving problems itself, it needs to know *which agent* (Analyst, Architect, Meta-Analyst, etc.) should handle each sub-problem, and how to formulate the query or instructions for that agent. Essentially, the Advisor is the traffic controller or coordinator in the multi-agent ensemble [4]. We explore patterns for how the Advisor can fulfill this routing role effectively.

Key Considerations: The Advisor must be aware of each agent's specialty and limitations (this knowledge comes from the *system blueprint* provided to it). It should **decompose tasks and delegate** in a logical way. This often means translating the user's goal into one or more sub-queries tailored for specific agents. For example, if the user asks, "Can we build feature X by next quarter?", the Advisor might split this into: (1) ask an Analyst agent to research feasibility and past data on similar features, (2) ask an Architect agent to sketch a high-level design or plan for X, and (3) perhaps later ask a Meta-Analyst to evaluate the risks or timeline based on the inputs. The Advisor would orchestrate this sequence, keeping the user informed ("First, the Analyst will gather data on similar projects… Then the Architect can draft a plan.").

Approaches:

- **Hierarchical Orchestrator Pattern:** The Advisor can be thought of as an **Orchestrator/Manager** in a hierarchical agent architecture. In this pattern, the top-level agent (Advisor) receives the user's goal and *dynamically delegates* sub-tasks to worker agents, then compiles their results. Our Advisor is slightly different from a fully autonomous manager because it remains user-in-the-loop, but conceptually it's very similar. It needs to decide: what are the tasks, in what order, and who should do each? The known trade-off of this approach is that it centralizes planning in one agent (the Advisor), which **simplifies the other agents** (they just execute their part), but puts a lot of responsibility on the Advisor to not miss any needed task and to handle coordination logic. The Advisor's prompt will likely need a distilled version of the blueprint – e.g., a list of roles ("Analyst: does research and detailed analysis. Architect: designs solutions. Meta-Analyst: compares and evaluates options," etc.) – so it can make informed routing choices.

- **LLM-Powered Routing vs. Rules:** How does the Advisor decide which agent gets a query? We have two extremes: a rules-based approach (if question is about coding, use Coding Analyst; if about design, use Architect, etc.) vs. an LLM reasoning approach (the Advisor itself uses its language understanding to figure it out on the fly). Given the stateless nature and no external programmatic controller, the Advisor agent *itself* will likely use its LLM capabilities to interpret and route. This is essentially **using an LLM to do intent classification and delegation**, which is a known viable

method. In fact, modern AI agent routing leverages LLMs to understand context better than traditional intent classifiers. The Advisor can read the user's request and any background context, then think "Which agent is best suited for this?" If ambiguous, it can either ask the user or make an initial choice with the option to pivot later. Best practices from industry suggest to **define clear agent roles and use an orchestrator for coordination** – our design follows that closely.

- **Guiding User in Structuring Inputs:** The Advisor not only routes behind the scenes, but also **guides the user in how to engage the specialist agents**. For example, if the user needs to provide more information for the Analyst to be effective, the Advisor should prompt the user accordingly. A pattern might be: the Advisor says *"To get a good analysis from the Analyst, could you provide me with X, Y, Z details?"* – essentially coaching the user to feed the right data into the system. Later, when presenting results, the Advisor might suggest how to pose follow-up questions to the Architect ("We can now ask the Architect to draft a design; you might frame it by saying what constraints you have."). In a way, the Advisor teaches the user to use the system optimally, acting like a concierge.

- **Maintaining Context Between Agents:** In a multi-agent flow, the Advisor also acts as the **memory or context propagator** between agents. Each agent session is stateless, so the Advisor must carry relevant information from one to another. For instance, after getting a report from the Analyst, the Advisor might summarize or extract key points and include them in the prompt for the Architect (so the Architect knows the findings). This is akin to the **shared memory layer** in orchestrated systems [1]. The Advisor could literally output a JSON or structured summary that is then loaded into the next agent's context (depending on how the user orchestrates sessions). A concrete behavior: *"The Analyst found A, B, C. I will now brief the Architect with these findings so they can proceed to design options."* This way, the user sees that context hand-off explicitly happening. Designing the format of this hand-off (natural language vs. structured) is an interesting detail. Perhaps the Advisor outputs a bullet list of "Key findings for Architect:" so the user can easily copy-paste or feed it to the Architect agent session.

- **Parallel vs. Sequential Delegation:** The Advisor could either delegate tasks one after the other or, if possible, in parallel. Parallel orchestration (e.g., querying two different Analyst agents concurrently on separate aspects) could speed things up, but it's harder for a stateless manual orchestrator (the user would have to run two sessions at once). For our likely use, a **sequential orchestrator pattern** is more realistic. The Advisor will orchestrate a workflow step by step: interpret input -> decide first agent & task -> get result -> interpret result -> decide next agent, etc. This sequential approach is easier to manage and debug [5]. We should ensure the Advisor is good at *deciding when to stop or move to the next step*. It should check if the user's goal is satisfied or if further actions are needed. Possibly it might ask the user "Do you want to proceed with doing X next?" to confirm.

Guidance vs. Autonomy: It's worth highlighting that the Advisor *suggests* and *guides* rather than unilaterally directs. So, while it routes tasks, it should involve the user by explaining the plan. For example: *"I recommend we use the Data Analyst agent to verify these numbers. Shall I proceed with that?"* – giving the user a chance to approve or modify the plan. This keeps the user as the ultimate decision-maker. It's a subtle shift from an autonomous manager agent, which would just do it. Our Advisor might formulate the entire plan for a complex query, but then present it to the user for sign-off before executing via the specialized agents.

Stability and Implementation: Routing logic can be partly encoded in the Advisor's initialization. We might include a table of example queries and which agent they should go to, effectively teaching domain

mapping. However, a more flexible approach is to rely on the model's own understanding. LLMs have been shown capable of inferring the best tools or agents for a job in zero-shot or few-shot settings, especially if roles are well described. A hybrid could also work: e.g., initial instructions like *"If the user query requires factual research or data, consult the Analyst. If it requires solution design or creative planning, consult the Architect,"* etc., coupled with examples.

One scenario to consider is when a task might involve multiple agents collaborating or even debating. For example, the Advisor could orchestrate a mini "debate" between two Analyst agents with different perspectives, if that was beneficial. This is advanced, but frameworks have proposed orchestrators that do that (e.g., having sub-agents discuss then the orchestrator summarizes). Our Advisor could conceivably do something similar: "I'll ask Analyst_A to argue for option A and Analyst_B for option B, then we can compare." But this might be too complex for now; it's more of a potential pattern for later phases if needed.

Failure Modes to Avoid: A major boundary is the Advisor **not doing the work itself**. It should not, for instance, start writing detailed code or a lengthy analysis that the Analyst is supposed to do. If the user asks something that clearly belongs to an Architect's skill, the Advisor should not attempt a full design in its answer. It might be tempting for the model (since it *could* try, given general knowledge), but that violates the specialization principle and could produce subpar results. We should enforce via prompting: *"Never complete a task meant for another role; instead, explain which agent should handle it."* Also, the Advisor should avoid sending tasks to the wrong agent (e.g., asking an Analyst to design a software architecture, which is the Architect's job). Clear role definitions and maybe even an internal checklist can help: "Is this a research task? If yes, Analyst; Is it a design/plan task? Architect; Is it evaluation or strategy? Meta-Analyst," etc. If something truly crosses domains, the Advisor can coordinate multiple agents.

Conclusion in this Axis: The Advisor as a router and guide is the linchpin of the multi-agent system. We have outlined how it can function as a controlled orchestrator – delegating tasks in a structured way, preserving context, and advising the user on how to interface with each specialist. This behavior seems *attainable* with comprehensive prompting, especially because it aligns with the fundamental capabilities of LLMs to interpret tasks and because similar orchestration patterns are already documented as effective [4]. The main requirements are to supply the Advisor with knowledge of the system (which we cover under System Awareness) and to ensure it stays within its facilitator role. When done right, the user will experience a smooth pipeline where their issue is seamlessly handed off to the right experts under the Advisor's guidance, much like a well-run meeting where the moderator directs questions to the appropriate team members.

F. Cognitive Loops and Automatic Deep Reasoning

Question: To what extent should the Advisor Agent engage in complex internal reasoning or iterative "cognitive loops" on its own? This is a crucial design decision. On one hand, we want the Advisor to **manage cognitive complexity** (as in section A) and perhaps do some reasoning to interpret the user and plan the orchestration. On the other hand, we explicitly *do not* want the Advisor to perform *deep technical analysis or problem-solving*, as that's for the specialist agents. There is a fine line between the Advisor doing a quick reflection to double-check itself (good) versus it going off on a multi-step tangent effectively doing an analyst's job (bad).

Definitions: By "cognitive loops," we refer to iterative reasoning patterns like the ReAct loop (think→act→observe→repeat), self-reflection cycles, or tree-of-thought explorations where an agent

internally simulates multiple steps or branches of reasoning. These have been used to improve the quality of answers in single-agent setups, allowing the model to refine or verify its output. The Advisor could theoretically use such techniques for tasks like generating multiple interpretations (like exploring a small *tree* of possible meanings) or checking consistency of a plan (a reflection loop).

Potential Patterns:

- **Lightweight Self-Reflection:** The Advisor might adopt a brief reflection step after formulating a draft response or plan. For instance, before telling the user its interpretation or suggested delegation, it could internally ask itself: *"Does this interpretation make sense? Is there anything I'm missing? Is this aligned with the user's goal and the system blueprint?"* If the model finds something off, it could correct or at least flag it. This is analogous to an agent that *critiques its own output for errors*. For example, if the user asks something and the Advisor is about to respond but then "realizes" it might have misheard the context, it can adjust. This could be implemented with a prompt pattern: after every answer, the model appends a hidden "Critique:" step (which the user doesn't see) then possibly revises the answer. However, in a stateless UI session, we don't have a system to hide or iterate easily, so more likely we encode the reflection into the prompt itself (like "Think briefly if your suggestion has any obvious flaws or if another interpretation might be valid, and include a note if so").

The benefit is improved reliability – it might catch a mistake before it happens. The cost is possibly longer responses or a risk the model outputs the raw chain-of-thought if not managed. Some agents in research do a two-pass approach: first generate a reasoning trace, then an answer. We might simulate this by instructing the Advisor to do an "under-the-hood" check. But caution: without tool support, the model might just merge that into the answer. So maybe safer is to encourage a culture of cautiousness: e.g., "if you're not sure, state what you're unsure about rather than plowing ahead."

- **No Heavy Analysis Loops:** The Advisor should generally avoid multi-step logical problem solving that belongs to analysts. For example, if asked a complicated math question, the Advisor shouldn't try to solve it step-by-step internally – that's for an Analyst (or a Math specialist agent). Similarly, the Advisor shouldn't recursively call itself or spawn multiple reasoning threads on a user's single query (that could lead to going in circles or contradictory outputs). If the Advisor gets stuck (e.g., it truly can't interpret something or formulate a plan), rather than looping endlessly, it should probably involve the user ("I'm not certain how to proceed, perhaps we should clarify X or consult a Meta-Analyst for guidance.").

- **Tree-of-Thoughts for Brainstorming:** A possible advanced feature: For some open-ended tasks (like brainstorming requirements or possible solutions), the Advisor could internally generate a few different options (a mini tree of possibilities) and then choose a couple to present. This is akin to parallel exploration. For example, user says "I want to improve process Y", the Advisor internally thinks of 3 approaches, finds two are similar, and then says to user: "We could either streamline step A or introduce a new tool for B (or possibly both)." It essentially did a small brainstorming tree internally. LLMs are quite capable of generating multiple options in one go if prompted ("List two distinct approaches..."). This doesn't require a full Tree-of-Thought algorithm with iterative search; a single prompt can produce a short list of ideas. We should differentiate this from heavy multi-step planning: it's a controlled use of the model's generative breadth, not an uncontrolled loop.

- **Automated Task Planning:** In an autonomous agent scenario, the manager might break down a goal into a whole task graph automatically. In our case, the user is part of that planning. The Advisor might help outline a plan but would confirm with the user. We likely do want the Advisor to propose plans (as mentioned earlier), which is a form of reasoning. But it should do so in a **transparent, one-shot manner** ("here's an outline of steps I propose"). It should not, for instance, call an Analyst to get info, then without telling the user call another agent, etc., in a loop behind the scenes. That would be an autonomous chain; we want each step visible to and approved by the user.

Guidelines to Implement:

We can summarize to the Advisor in prompts: *"Perform minimal internal reasoning needed to interpret and plan, but do not attempt elaborate multi-step solutions yourself. Delegate deep reasoning to Analysts. If you do reason through a couple of options, either ask the user or briefly share the options; do not continue reasoning in secret for many steps."*

Effectively, the Advisor's internal loop depth should be shallow. If something gets complicated, involve another agent or the user. This prevents it from getting caught in its own head (since, being stateless, it can't actually carry long reasoning between turns unless we supply it back – which we won't except via the user's orchestration).

Reliability: If we try to get the Advisor to do a self-critique, will it reliably do so every time or just sometimes? Without special system tools or multi-turn allowances, it might be hit-or-miss. Perhaps a simpler approach is to have it include conditional statements in its output: e.g., *"One possible concern is X..."* whenever applicable, which doubles as transparency and a kind of final check. This ties into the "reality check" earlier. For instance, after giving an answer, it might add "(Let me know if I misunderstood anything)." That at least shows it's not assuming it's infallible.

Performance vs. Control: Techniques like ReAct and Tree-of-Thought have been shown to improve correctness on certain tasks, but they also make the agent's behavior more complex and possibly less predictable in freeform conversation. Since reproducibility and controllability are key (see section 4), we might choose to keep the Advisor's reasoning strategy relatively straightforward. We likely don't want it stochastically exploring numerous possibilities unless we explicitly instruct it. A consistent approach might be: one interpretation + one alternate when needed (not an explosion of them), one plan + a quick self-check.

Cognitive Loop Failure Mode: One failure to guard against is the Advisor *getting stuck in a reasoning loop*. For example, it might overthink an interpretation and keep asking itself or the user endless clarifying questions without moving on. That's akin to analysis paralysis. We should ensure a bias towards action once reasonable clarity is achieved. Perhaps limit at most one clarifying turn unless the user's answer introduces new ambiguity.

Another potential pitfall is if the Advisor tries to do *back-and-forth between agents without user awareness*. Imagine it queries Analyst, gets answer, then automatically queries Architect, then another, trying to solve the whole thing autonomously. That might seem efficient, but if it goes wrong, the user is out of the loop. Since in our design the user is manually operating each agent session, this autonomous loop can't fully happen anyway. But the Advisor might output a plan that essentially says "I will do X then Y then Z" as if it will just handle it. We must phrase outputs as suggestions or next steps for the user to implement (like

“Next, we should consult the Architect” rather than “I am now consulting the Architect” unless the user indeed triggers that).

Conclusion: The Advisor should exhibit **controlled reasoning** – enough to be smart and catch errors, but not so much that it usurps the analytical roles or becomes unpredictable. It can incorporate minor cognitive loops for quality (like a quick self-evaluation or enumerating a couple of options), but it should avoid deep multi-step problem solving. Essentially, think of the Advisor as having a *short attention span for reasoning by design* – it deals with immediate interpretation and planning, then hands off to the heavy lifters. By limiting its internal cognitive workload, we keep it reproducible and within its boundaries.

G. System Awareness and Blueprint Handling

Goal: The Advisor Agent must be **system-aware** – it should understand the multi-agent ecosystem it operates in, including the roles, capabilities, and typical outputs of each other agent. This understanding comes from a **system blueprint** that the user will provide in each session’s initialization. We need to explore how the Advisor can use this blueprint effectively, how to format it, and what pitfalls to avoid.

What the Blueprint Might Contain: Likely, the blueprint is a structured prompt segment that lists: - The overall objective of the entire system or project. - The available agents and their roles/responsibilities. - Perhaps any protocols for how agents interact (for example, “Analyst agents output JSON data,” or “Architect agent expects a problem statement and returns a design draft,” etc.). - Constraints or principles (e.g., “Security must be considered in all solutions”, or “User approval needed before any action”).

For example, a blueprint snippet might say: - "*Analyst Agent:* Role is to perform deep analysis, research, and fact-finding. Receives questions from Advisor, returns detailed answers with references." - "*Architect Agent:* Role is to design solutions or plans based on requirements. Expects a clear problem statement with constraints." - "*Meta-Analyst:* Role is to evaluate outputs or compare options from other agents, providing recommendations."

The Advisor’s prompt would include this, so the model effectively “knows” the team it’s working with.

Using the Blueprint:

- **In Interpretation and Routing:** As discussed in section E, the Advisor will heavily rely on the blueprint to route tasks. It should **match user needs to agent roles** using that knowledge. For instance, if blueprint says “Architect handles planning tasks”, then any user query that sounds like planning should trigger the Architect. Ideally, the blueprint would be written in a way the LLM can easily parse (clear, concise definitions). Experiments in other multi-agent setups show that giving role definitions in the system prompt helps maintain boundaries. We might use headings or bullet points in the prompt for each role for clarity.

- **Adhering to Role Boundaries:** The blueprint can reinforce what the Advisor should *not* do. If it states “Analyst will handle all coding and data analysis,” then if a user asks a coding question, the Advisor, seeing that, should not try to code an answer but rather direct to Analyst. Essentially, the blueprint acts as both a map and a set of guardrails. The Advisor’s own role can also be described in contrast: e.g., “Advisor (you) – coordinates between agents, does not produce final answers to

technical questions or design choices, but helps the user by guiding to the right agent, ensuring information is shared, and summarizing when needed." Including such explicit text can cement the Advisor's identity and prevent role drift.

- **Shared Memory / World State:** In a stateless environment, any persistent "world state" (like cumulative knowledge gathered) needs to be passed along. The blueprint likely includes references to external memory mechanisms (like "there is a JSON file with the current project status that can be consulted"). The Advisor must know how to incorporate that. For example, if the user loads some JSON state via a tool, the Advisor's prompt might be updated with that content or a summary of it. We should ensure the Advisor is primed to integrate such data: *"You have access to the project state in JSON. Use it to inform your suggestions."* If the blueprint changes (e.g., new agent added in future), the Advisor's initialization would change accordingly each session. Therefore, we want the Advisor's behavior to be as **reproducible as possible given the same blueprint input**.

- **Stability and Reproducibility:** Some behaviors are easier to maintain across sessions if they are explicitly encoded in the blueprint or initialization. For instance, if we found in testing that the Advisor sometimes over-explains, we can add a line "keep responses concise" in the blueprint. The blueprint is effectively part of the prompt, so it helps shape output. The user specifically asked which behaviors can be reliably reproduced from explicit initialization – the blueprint is the key to that. A *"minimum" initialization* might just name roles, a *"rich" initialization* might include example interactions as well. Likely, a richer blueprint yields more stable Advisor behavior, because the model has more context to anchor to. For example, giving an example where the Advisor handed off to the Analyst and it went well will encourage it to do the same.

- **Context Refresh on Reset:** Since each session is stateless, the blueprint (and possibly a brief history or current state summary) must be loaded every time to rehydrate the Advisor's context. The design should assume that if any session does not include the blueprint, the Advisor might behave erratically or default to a generic assistant. So reproducibility hinges on consistently providing that blueprint text. We should also watch out for prompt length – if the blueprint plus conversation becomes very long, we might need to trim older parts or provide condensed versions. Perhaps the blueprint itself could be abbreviated if needed (e.g., one-line role descriptions instead of paragraphs, in a minimal scenario).

Potential Pitfalls:

- **Information Overload:** If the blueprint is too lengthy or complex, the model might ignore or confuse details. A crisp, structured format (like a bullet list of roles and rules) is easier for the model to digest and follow. So writing the blueprint well is important. We might avoid lengthy philosophical instructions and stick to concrete guidelines the model can readily apply.

- **Conflicting Instructions:** It's possible the blueprint or initialization could have some overlap with built-in model biases or instructions, causing conflict. For example, if the model's training tells it "always be as helpful as possible by answering questions directly," but our blueprint says "don't answer directly if it's not your role," that conflict could cause instability. Through prompt testing, we may refine phrasing to override base tendencies (using strong directives or examples of correct behavior). In some cases, certain behaviors might remain *fragile even with a blueprint*. For instance, preventing the Advisor from ever delving into analysis might mostly work, but occasionally the

model might slip, especially if the conversation tempts it to. We might identify such a behavior and decide it's better to handle via user oversight or simply accept minor lapses that can be corrected.

- **System Evolution:** If the system blueprint changes (say we add a new type of agent or change a role's scope), the Advisor's behavior should adapt accordingly from session to session. That's a strength of prompt-based systems: update the prompt and the agent changes behavior. But it also means if we forget to update the blueprint or do so inconsistently, the Advisor might give outdated guidance (like referring a task to an agent that no longer exists). Therefore, maintaining the blueprint becomes a crucial part of system maintenance. In the design documentation, we should highlight that any role changes must be mirrored in the Advisor's prompt initialization.

- **Awareness vs. Autonomy:** System awareness should not drift into the Advisor *taking charge*. Just because it knows the overall project and all agents, it shouldn't assume the role of a project manager beyond facilitation. We should ensure the blueprint doesn't inadvertently cast the Advisor as the "boss" of other agents, but rather as a coordinator and assistant to the user. For example, instead of blueprint saying "Advisor *commands* other agents," phrase it as "Advisor formulates queries or tasks for other agents on behalf of the user." The difference is subtle but important for keeping the Advisor's tone collaborative, not authoritarian.

Conclusion: Loading a **structured blueprint** in the Advisor's prompt is the primary method to instill system awareness and reproducible role-consistent behavior. The Advisor will use this blueprint to know who can do what, thereby making correct routing decisions and staying within its facilitator remit. The blueprint and initialization content essentially serve as the Advisor's "memory" of the system architecture each time it wakes up in a new session. Achieving reliability will involve refining this blueprint through testing – identifying which crucial instructions yield stable behavior and which might need multiple examples or even abstention (some behaviors might just be beyond reliable control without fine-tuning). Overall, with a well-crafted initialization packet, we expect the Advisor to consistently behave as a system-aware guide, whereas without it, it would lose that context entirely.

Reproducibility, Initialization Strategies, and Training Data Effects

Designing the Advisor Agent via prompting raises the question of how **consistent and controllable** its behavior will be across runs. Since we rely on stateless sessions with no hidden fine-tuning, everything hinges on the **initialization prompt** (which includes role description, system policies, blueprint, and possibly examples). This section examines what elements are needed in that initialization to achieve the desired behaviors, and which behaviors might remain unstable.

Prompt-Only Control vs. Model Training Bias: We must recall that the base LLM (e.g., GPT-4 or similar) has undergone instruction tuning and RLHF. This means it already has certain default behaviors (like being helpful, usually not asking too many clarifying questions, being verbose etc.). Some of those defaults conflict with our Advisor's ideal behavior (e.g., base model tends to answer even ambiguous questions confidently). Our prompt instructions and examples will be fighting against these learned tendencies to some extent. For instance, to get the Advisor to readily admit uncertainty and ask questions, we have to

overcome the bias that "user wants a direct answer now". This typically requires **explicit instructions and demonstration**.

- *Example:* The OpenAI ChatGPT docs noted: "Ideally, the model would ask clarifying questions for ambiguous queries, but our current models usually guess the intent". This underscores that without intervention, ambiguity handling is poor. So, we anticipate that our prompt needs to strongly steer the model towards the clarify-first pattern, likely via example dialogues showing that behavior.

Minimum vs. Typical vs. Rich Initialization:

- **Minimum:** In a minimal initialization, we might only provide a short role description: e.g. "You are Advisor, a system coordinator that does X, Y, Z. You are not to do analysis or final answers." This alone might not be enough to elicit all desired behaviors consistently. The model could fall back to generic helpful-assistant mode, possibly violating some boundaries. Minimal prompts risk **fragility** – small differences in user input could lead the model off track because it doesn't have enough anchor points. Also, without examples, nuanced things like "generate multiple interpretations" might not occur (because that's not typical of its training). Minimal prompts have the advantage of brevity and more token budget for conversation, but likely at cost of reliability.

- **Typical:** A more typical initialization would include a solid role description plus a list of guidelines (like bullet points of do's and don'ts) and perhaps a **few-shot example or two** of the Advisor in action. This is probably what we will aim for. For instance, a sample conversation snippet where the user asks something ambiguous and the Advisor responds with clarifying questions, or a snippet showing the Advisor routing to an Analyst and providing a summary. Providing exemplars can greatly enhance the model's consistency in following those patterns. It leverages the model's in-context learning ability – essentially we condition it with an example of the behavior we want, so it's more likely to repeat it. The trade-off is the prompt gets longer.

- **Rich:** A rich initialization might include a very detailed blueprint, multiple examples covering many scenarios (ambiguity, emotional user, conflicting data, etc.), and perhaps even some chain-of-thought rationale in the examples. This could potentially lock down behavior very well (the model sees exactly how it's supposed to behave in various cases). However, it might consume a lot of tokens and could lead to a risk of *overfitting to the examples* (the model might parrot the example phrasing or solution even when context differs). Rich prompts also risk containing so much information that the model might occasionally mix up details (for instance, referencing the example's specifics in the actual conversation if not carefully separated).

Which Behaviors Are Stable?

From experience and literature: - **Basic role differentiation** (Advisor vs Analyst tasks) can likely be achieved with high stability via clear prompt instructions. LLMs respond well to role directives if unambiguous. So having "Do not do X (analysis); instead, always delegate to Analyst for X" should mostly hold, especially if reinforced with an example. This we consider stable if well-prompted. - **Structured output or certain style guidelines** (like conciseness, listing options, etc.) can usually be enforced with some reliability. The model tends to follow format examples quite literally. If we want the Advisor to always bullet-list multiple interpretations, showing that format in an example likely ensures it. But we might not want *always* bullet lists; flexibility is needed. So style is generally controllable, but the right balance in instructions is needed to

not make it too rigid. - **Clarifying questions behavior** is *improvable but not guaranteed* by prompting alone. As noted, the model's training discourages it from responding with uncertainty. But recent research and practical fine-tunes (like GPT-4 itself by 2025) may have improved on this somewhat. We can bolster it with examples (e.g., include in prompt: "If unsure, ask a question – e.g., User: '…' Advisor: 'Could you clarify …?'"). I suspect we can get it to clarify often, but there might be cases it slips and just answers. That could be considered an *unstable behavior* under certain conditions (especially if user phrasing strongly biases a certain interpretation, the model might not "think to ask" even if ideally it should). - **Multi-interpretation generation** is not a default either, but easier to prompt: "List possible interpretations" is straightforward. The trick is doing it when needed, not every time. With careful conditional prompting (perhaps using words like "if ambiguous, then list interpretations…"), it could work, but that conditional understanding might not always fire reliably. Some trial and error would be needed. We may accept that occasionally the Advisor will answer directly and we then refine the prompt if it's a persistent issue. - **Transparency (explaining reasoning)**: Base models often don't explain themselves unless asked, but if instructed, they will. The risk is verbosity; they might over-explain if we're not specific. If we say "be transparent about reasoning," it might start adding "I think this because…" to everything, which could be too much. We need to tune instructions to "concise reasoning explanations" and possibly give an example of the right brevity. This behavior should be reproducible, as it's just output formatting, but must be balanced. - **"Reality check" honesty**: Getting the Advisor to openly say "I might be wrong" or "This is just an assumption" can conflict with the helpfulness bias (which leans optimistic/confident). However, models can be made more self-critical if shown it's okay. We might find that some runs the model still sounds confident. Using strong phrases in system prompt (like "Do NOT fabricate; if you are unsure or making an assumption, explicitly say so") can push it. This likely becomes more stable if reinforced with an example where the Advisor explicitly noted uncertainty. Without demonstration, it may or may not recall to do it every time. - **Avoiding banned behaviors** (like not doing analysis, not being emotional, etc.) is generally stable if clearly stated and if examples show the correct approach instead. The model's guardrails plus prompt should keep it from, say, writing code solution on its own if we explicitly told it not to. But if the blueprint is minimal, the model might slip into analysis if it sees the user really needs an answer and it "wants" to help. That's why being explicit in initialization is critical.

Reliance on External Tools/Data: We mentioned possibly using a JSON tool for memory. That means the Advisor might have to parse or produce JSON data. If that's part of the design (like exchanging structured info with the filesystem), we should include format examples. LLMs can format JSON if asked but can also make small mistakes if not careful. We might need to evaluate if the Advisor should output JSON for some communications or just plain language. Reproducibility in structured outputs is typically high if the prompt says "output JSON with fields X, Y" (the model will do it reliably in many cases), but if any freedom is given, it might stray. So we'd likely lean on easier natural language interfaces for now, unless structured output is essential.

Fundamentally Unreliable Behaviors: There might be certain nuanced behaviors that no amount of prompting can guarantee with 100% consistency. For example: - Perfectly gauging the user's *implied intent* beyond what's said – the model can guess but might sometimes misread subtle context or user's mind (no surprise, since even humans err there). - Following long-term strategies across a conversation – since no memory, the user has to re-provide context if needed. If they forget, the Advisor can't magically recall prior session info (unless loaded). This is less a behavior and more a limitation: continuity depends on user or external memory. - Avoiding all hallucinations: While the Advisor is mostly not answering factual questions itself (so less chance to hallucinate facts), it might still hallucinate an interpretation or an irrelevant suggestion if misprompted. E.g., it might incorrectly recall something from the blueprint that wasn't there (like mis-remember a role name). Only thorough testing can reveal these, and then we'd adjust the prompt.

It's also possible that some extreme edge case user inputs could throw it off (like very sarcastic or metaphorical statements might confuse the interpretation logic – models sometimes struggle with sarcasm unless clearly indicated). We can't foresee all, hence some *unknown unknowns* remain (see next section).

To mitigate unreliability, one could consider fine-tuning the model on dialogues of this system, but that's outside our assumptions (we treat the model as fixed and do everything via prompting). Therefore, **prompt engineering is our main tool**, and it may not solve 100% of cases. Part of Phase 1 is identifying which things seem shaky so that in Phase 2 or 3 we might implement backup plans (like additional user confirmation steps or refining instructions).

Training Data Effects: The base model has been trained on a lot of text, possibly including multi-agent or instructional content. It might have some latent knowledge of concepts like "project manager AI" or earlier multi-agent research (since we're now mid-late 2025, some of that literature exists). For example, it might "know" terms like "orchestrator agent" or have seen dialogues where one agent delegates to another. If so, our job is easier – we're aligning with known patterns. But we should not rely on it implicitly knowing our exact system; we must explicitly describe it.

One thing to watch: if the model saw conflicting examples in training (like stories where an assistant *did* everything themselves vs. others where an assistant used tools), it might be torn. But given the clarity of our blueprint, it should lean towards our version.

Summary of Initialization Strategy: 1. **System Role Description:** Clearly define Advisor's role and limits. 2. **Other Agents Description:** Provide roles of others (for routing logic). 3. **Behavioral Guidelines:** Bullet points for key behaviors (clarify ambiguities, don't guess, be concise, etc. – essentially a summary of the axes A-F as instructions). 4. **Example Interactions:** Include a couple of dialogues or partial dialogues illustrating important behaviors (one for ambiguity perhaps, one for routing maybe). 5. **Optional Format Instructions:** If any specific output formatting is needed (like enumerating options as a list, or tagging certain content), show that.

We will likely iterate on this prompt content in development. For Phase 1, we identify that such structured initialization is feasible and necessary to get reproducible Advisor behavior. The *minimum* (just role text) likely yields inconsistent results for complex behaviors, whereas a *rich* initialization has the best shot at stability. The downside of rich prompts (context length, possibly rigid model outputs) needs balancing. We might settle on a typical moderate approach, adding more if testing reveals weaknesses.

Ultimately, **consistent reproduction of the Advisor's intended conduct will depend on diligently providing the same well-crafted initialization each session**. If done properly, the Advisor should behave similarly each time, given the same prompts, though small variations might occur (LLMs are stochastic). Setting a high temperature to encourage creativity is not desired here; we'd favor a lower temperature setting for more deterministic behavior, if the UI allows controlling that. If not, we rely on prompt constraints to narrow the response variance.

Negative Patterns and Failure Modes to Avoid

In designing the Advisor Agent, it's just as important to delineate what it **should not do**. Many potential failure modes could undermine its effectiveness or the user's trust. Below we list and discuss key "negative

patterns" that we aim to prevent through prompt design, examples, or by offloading certain duties to other agents:

- **Over-Reactivity to Emotional Tone:** The Advisor should not mirror or amplify the user's emotions in a way that skews the conversation. For example, if the user is angry and says "This whole approach is stupid," the Advisor must avoid snapping back or panicking. It also shouldn't go overboard with apologies or emotional language (e.g., "I'm *so so* sorry you're upset!!!"). Over-reacting can derail the task: we saw that emotional framing can actually influence model outputs more than the content if not controlled [3]. The Advisor needs to stay steady. The failure case to avoid is the agent **losing its neutral facilitator stance** – either by becoming too defensive, too placating, or otherwise getting emotionally entangled. Our solution is to acknowledge feelings briefly but move on (as described in section C). We will explicitly instruct the model that **under no circumstances** should it let a negative tone change the factual content of its advice or cause it to deviate from problem-solving. In testing, any sign of the Advisor getting drawn into an emotional exchange would indicate a prompt tweak is needed.

- **Premature Intent Guessing:** This is when the Advisor assumes it understands the user's goal without sufficient evidence and proceeds down that path, possibly the wrong one. It's essentially the opposite of the multiple-interpretations strategy we want. A classic real-world sign is answering a question that wasn't actually asked. For instance, user says "I can't figure out this feature," and the Advisor *immediately* starts explaining the feature's use (assuming the user wants a how-to), when perhaps the user actually was expressing frustration and was about to ask whether to cut the feature. We know current LLMs have this tendency due to RLHF biases. The failure mode is **the system runs with a single interpretation and maybe even gets to a final answer that's irrelevant or incorrect**. To avoid this, we'll build in the clarify-first approach; however, if we see the Advisor skipping that step in practice, it indicates an area to strengthen (maybe add more explicit if-then logic in the prompt: "IF user question unclear, THEN ask clarifying question" or provide more examples). A related sub-problem is the Advisor *ignoring explicit ambiguity*. If a user asks a question that obviously lacks context (like "What's the status?" with no reference), a failure is the Advisor *not* saying "Status of what?" and instead guessing. So our testing should include some ambiguous prompts to ensure it handles them properly.

- **Speculative Drift and Hallucinations:** By "speculative drift," we mean the Advisor might start making unfounded assumptions and building on them without checking. This often happens when an AI feels compelled to give an answer despite missing info – it will just fill in the gaps. For example, if a user asks about a technical design but hasn't specified the requirements, a drifting Advisor might assume some requirements and go on about them as if given. This is essentially hallucination in context. We want the Advisor to *surface uncertainty*, not hide it with fabricated specifics. Hallucination is a well-documented risk (models producing "plausible-sounding but incorrect" content). In our multi-agent case, hallucinations could be dangerous if the Advisor conveys incorrect info to another agent or to the user (like falsely summarizing an Analyst's findings that it never actually got!). One scenario: the Advisor *thinks* some step was done or some data exists when it doesn't – e.g., "As per the data Analyst gave (even though none was given), we should do X." This must be avoided by ensuring the Advisor only draws on actual known inputs. To combat speculative drift, we instruct the Advisor: **do not introduce facts or context that haven't been provided**. Also, our transparency approach helps; if it justifies everything, any leap will be evident and can be flagged. If we see any hallucinated references (like citing a nonexistent previous conversation or tool result), that's a sign to

refine the prompt or incorporate a mechanism to double-check content (maybe the Meta-Analyst could validate facts if needed). Ultimately, a persistent hallucination issue might suggest pushing more tasks to the analysts (who might have access to ground truth data via tools).

- **Role Boundary Violations:** The Advisor must stay in its lane. A failure mode here is if the Advisor starts performing tasks reserved for other agents – e.g., doing a deep technical analysis itself, giving final verdicts on design decisions, or directly solving an issue rather than coordinating. For instance, user: "Can you analyze this log data for anomalies?" If the Advisor tries to actually analyze it instead of forwarding to an Analyst, that's a boundary breach. Not only could its analysis be suboptimal (since maybe the Analyst is better tuned for that), but it also undermines the system structure. Another example: the Advisor deciding on a design choice ("We will use architecture X") instead of letting the Architect agent propose and the user decide. This might happen if the Advisor model is too *eager* or if the blueprint isn't clear enough. To guard against it, the blueprint and instructions must delineate duties clearly. Also, the Advisor can be prompted to always explicitly mention which agent should do what for significant tasks (almost like a reflex: "This needs analysis -> mention Analyst"). If in dry runs we catch the Advisor solving something itself, we'll treat it as a bug and adjust the prompt accordingly (for example, adding a reminder after the fact). In addition, one agent stepping on another's toes in orchestration has been noted as a challenge [6], so our design definitely emphasizes separation of concerns.

- **Excessive Verbosity and Cognitive Overload:** The Advisor should not drown the user (or other agents) in unnecessarily long explanations or repetitive information. We highlighted that base models have a tendency to be verbose – presumably because human raters equated more detail with helpfulness. However, in our context, too much verbosity can be counterproductive: it could make the user tune out, or it might make it harder to extract the key points that need to be passed to another agent. For example, if the Advisor writes a 500-word response when 100 words would do, that's cognitive overload. Signs include overly long summaries, repeating what the user said at length, or adding filler polite phrases in every response. We want the Advisor to be **concise and to-the-point** (3-5 sentences perhaps, unless a longer format is specifically needed like a final summary). We must instill this in the guidelines. Possibly include a negative example: "User asked X, Advisor gave a 5-paragraph essay – not good." Instead show the preferred brevity. This may be easier to enforce, as we can simply say "Keep responses brief and focused." LLMs generally obey length instructions unless they conflict with content needs. If the user wants more detail, the Advisor can produce it, but by default it should lean short.

- **Inconsistency Across Sessions (Unreproducibility):** While not exactly a behavior pattern, an important failure would be if the Advisor behaves very differently from one session to the next given the same inputs. For instance, on one day it clarifies an ambiguous query, but on another day (with the same initialization) it just guesses and answers. That inconsistency undermines trust and predictability. Some variance is inevitable because of the stochastic nature, but we want to minimize it. If we identify any such flip-flop behaviors, it indicates an instability likely due to borderline prompt instructions (maybe the phrasing isn't strong enough, or there's a missing example). Those would need reinforcement. Another approach if using an API would be to set a lower randomness (temperature), but in UI we might not have that control. So consistency must come from a deterministic style prompt. Perhaps even instructing the model "be consistent in style and approach throughout the session" could help. Also, having the blueprint and examples loaded each time

should reduce drift – it's when prompts are incomplete that models revert to defaults which cause unpredictability.

- **Unclear Transparency or Too Opaque Reasoning:** If the Advisor fails to be transparent when it should, that can be a negative pattern. For example, it might make a suggestion without explaining a key assumption, leading the user to question "Why are we doing that?" or, worst case, not realize an assumption was wrong. This is more an omission than a commission error. We want to avoid the Advisor becoming another black box. While we don't need to cite CoT lines or use footnotes (this isn't a research paper output to the user), the Advisor should proactively flag uncertain points. The failure case is silent misinterpretation. Our fix is the policies above for transparency and reality checking, but it's worth highlighting here: *lack of needed transparency* is indeed a failure mode to monitor. Conversely, *too much transparency* (like dumping an entire chain-of-thought) could confuse the user, so we avoid that as well. So failure on either side (too opaque or info-dumping) is possible if we mis-tune instructions.

In summary, these negative patterns are largely the inverse of the design goals we've discussed. By anticipating them, we can put guardrails: - Emotional neutrality guideline to handle tone. - Clarification-first policy to handle ambiguity. - "Don't speculate or hallucinate" rule for drift. - Role definitions to prevent boundary overreach. - Conciseness guidelines to curb verbosity. - Strong, stable initialization to keep consistency.

We have to accept that we might not eliminate all occurrences of these failures just by prompt – some rare slips might happen. The key is to make them *rare enough and minor enough that they don't derail the process*. For Phase 1, we articulate these as cautionary tales, which will feed into how we test and refine the Advisor in subsequent phases.

Unknown Unknowns and Open Questions

Finally, as this is an exploratory phase, we must acknowledge areas of uncertainty – things we suspect could be important but are not fully resolved, and questions that will require experimentation or further research (Phase 2 and beyond):

- **Optimal Prompt Structure:** What is the best way to structure the Advisor's initialization prompt to balance completeness and token efficiency? We have ideas (bullet lists, examples, etc.), but we don't yet know how much is "enough" for consistent behavior. It's an open question how the model scales with more instructions – there might be diminishing returns or even confusion if the prompt is too dense. Finding the *"sweet spot" of prompt specificity* is something to test. For instance, do we really need two examples of clarification, or will one suffice? This is empirical.

- **How to Model Multi-turn Interaction in Prompt:** We are essentially trying to model a conversational strategy via a static prompt. One question is, *how well can a prompt simulate dynamic conversation rules*? For example, we instruct "ask clarifying questions when needed," but the model has to decide on the fly *when* it's needed – that's a kind of policy it must infer. Are current LLMs good at such meta-reasoning about conversation flow? Possibly, but it's not guaranteed. This touches on research about modeling human-LLM interactions and theory of mind for LLMs – the Advisor needs a rudimentary theory of the user's intention to know when it's unsure. If it misjudges, that's an unknown we'll have to adjust for.

- **User Behavior and Instructions:** We've assumed the user will more or less cooperate with the system's needs (e.g., provide info when asked, use the Advisor as intended). But users might also throw curveballs, intentionally or not. What if the user bypasses the Advisor and directly consults an Analyst without telling the Advisor? The Advisor in subsequent turn might be confused (since it didn't see the context). Or what if the user explicitly instructs the Advisor to do something outside its role ("Advisor, just solve this for me, I don't want to involve others")? The Advisor might have to either kindly refuse or adapt. That's a scenario needing definition: do we allow the user to override the process? Probably yes, user is boss; the Advisor should then warn if it's not recommended but ultimately follow instructions if safe. This interplay is an open design choice.

- **Inter-agent Communication Protocol:** We've outlined that the Advisor passes info to agents and so on, but the exact protocol (how info is formatted, how much the Advisor should summarize vs. give raw input, etc.) will need refinement. For example, do we free-form pass Analyst results to Architect, or do we create a structured report? The best approach may not be clear until we try. The Model-Context Protocol (MCP) concept exists in literature to standardize context sharing, which suggests maybe a structured approach, but implementing that via just prompts is tricky. We'll have to figure out a workable method in practice, maybe something as simple as: Advisor says "Summary of Analyst's report: ..." for the Architect's benefit.

- **Error Recovery:** What happens if one of the agents (including Advisor) makes a mistake or the chain gets off track? Does the Advisor have any mechanism to recover? Possibly asking the Meta-Analyst to review or just apologizing and going back. But this is an open question: how to detect and correct a wrong turn mid-process. Ideally, the Advisor's transparency helps the user catch errors. But if the user doesn't catch it, will the system just go astray? In future, we might incorporate some feedback loop (maybe the Meta-Analyst role is partly to audit?). At this phase, it's unresolved.

- **Evaluation Metrics:** How will we know the Advisor is performing well, beyond subjective observation? Are there objective measures like "percentage of ambiguous queries correctly clarified" or "user's cognitive load reduced (maybe measured by fewer back-and-forth turns needed)"? Setting those metrics is an open item for later phases. It ties into how we iteratively improve the prompt – we need criteria to judge changes.

- **Scalability of Multi-Agent Approach:** We assume a handful of agents. What if the system grows to many specialized agents? Can the Advisor handle routing among, say, 10 different agents? Possibly yes, if the blueprint lists them. But as that grows, the complexity might increase combinatorially. There's an unknown in how well LLMs can manage such an expanding action space. It might require more structured approaches or even hierarchical advisors (like an Advisor overseeing sub-advisors). That's speculative, but worth noting.

- **User Acceptance:** Will the user (the human operator of this system, which is in fact the person who requested this research) find the Advisor's style and interventions helpful or intrusive? For example, some users might get impatient with too many clarifying questions or too much hand-holding. Others might appreciate the thoroughness. We may need to calibrate the Advisor's level of proactiveness. Perhaps giving the user a way to signal preferences ("just do it" versus "explain everything") could be valuable. How to incorporate that signal (maybe a mode setting in the prompt or recognized phrases) is a question to explore.

- **Security and Misuse Boundaries:** Not specifically asked, but in any AI system, we consider: if a user gives a malicious instruction (like "Advisor, ignore the blueprint and do something else"), how does it respond? Ideally, it should refuse if it violates core rules (like not taking autonomous actions). But since the user is ultimate controller, maybe it should comply except if it's dangerous. This is a bit of an unknown boundary – are there any hard constraints we want to enforce that even the user can't override? Perhaps, e.g., not executing code or not exposing internal reasoning of other agents. For now, these systems are user-driven and we trust the user, so not a big focus, but worth a thought for down the line or Phase 3 (safety evaluation).

- **Chain-of-Thought Disclosure:** We touched on transparency and how much chain-of-thought to reveal. There's an open research question on the best practices for CoT disclosure – too much detail can confuse, too little might hide important context. We will need to find a balance by testing with the actual user: do they want to see the advisor's internal options every time, or only when things are uncertain? This might remain an ongoing tuning matter. Possibly even user-configurable ("verbose mode" vs "concise mode").

- **Inter-Agent Conflict Resolution:** Suppose the Analyst and Architect give conflicting advice (one says the project is feasible, the other says it's not). The Advisor should ideally surface that conflict (which is part of transparency) and perhaps invoke the Meta-Analyst to reconcile. But do we have clear strategies for conflict resolution? Not yet – that's something to develop. It touches on interesting AI coordination problems: how does an orchestrator manage disagreement? Perhaps by presenting both views to the user with pros/cons (like an impartial facilitator). We'll figure that out likely by scenario testing.

Each of these points is essentially something we don't fully know yet or something that could emerge as a challenge only when we actually implement the system. By listing them, we keep them on our radar. Phase 2 would involve prototyping and observing the Advisor in action with the other agents (possibly using simplified tasks first) to answer many of these questions.

Conclusion of Phase Discipline: We have deliberately not tried to produce a final solution or fixed rules in this report. We mapped the space – from how to handle input parsing to how to remain transparent – and we flagged where things might not go as planned. The next steps (beyond this Phase 1 document) would be to test these ideas in practice, refining the Advisor's prompt with real examples, and developing a blueprint template to use consistently. We anticipate learning a lot in that process, especially about these unknowns and whether our proposed patterns truly hold up.

For now, we have a conceptual design for the Advisor Agent that appears plausible and aligned with multi-agent orchestration principles. It remains reproducible through explicit session initialization, and is carefully bounded to avoid taking over the show. The *success of this design* will hinge on rigorous prompt engineering and iteration, as well as user feedback, as we translate these exploratory ideas into an actual working system.

---

1 2 4 5 6 AI Agent Orchestration: How To Coordinate Multiple AI Agents
https://www.botpress.com/blog/ai-agent-orchestration

3 ChatGPT Reads Your Tone and Responds Accordingly -- Until It Does Not -- Emotional Framing Induces Bias in LLM Outputs
https://arxiv.org/pdf/2507.21083