ChatGPT

LLM Workflows/Stateflow Service – Technical Brief

Authored by Dinis Cruz and ChatGPT Deep Research

Introduction

The **LLM Workflows/Stateflow Service** is a proposed stateless web service for executing AI-driven workflows with well-defined, deterministic steps. It acts as a **state machine** for orchestrating Large Language Model (LLM) calls and other actions in a controlled sequence. Unlike “agentic” AI systems that freely decide each action, this service uses a **pre-defined flow (blueprint)** to ensure predictability and provenance. The goal is to harness the power of LLMs (for analysis or content generation) within strict boundaries: the LLM can perform complex subtasks, but it *never* dictates the workflow’s overall path [1]. This design follows the *“Blueprint First, Model Second”* principle, decoupling high-level logic from the probabilistic nature of LLMs [1]. The result is an execution environment where every step is explicit and auditable, yielding reliable behavior even in complex multi-step AI tasks.

Key collaborators: The creation of this service is a collaborative effort between human developers and AI assistants (LLMs). Developers will define the workflow blueprints, implement the engine, and integrate services, while advanced LLMs (such as ChatGPT) assist in brainstorming, research, and even code generation. For example, this very document is co-authored by Dinis Cruz and ChatGPT, reflecting the human-AI partnership in designing the system. By leveraging LLM support during development, the team can explore design options and documentation rapidly, then validate and refine them with human expertise. This collaboration ensures the final system design is both innovative and technically sound.

Design Principles and Goals

- **Deterministic Control Flow:** All workflows are executed according to a fixed, predefined sequence of steps (a *stateflow*). The LLM is not allowed to randomly alter the flow or insert unforeseen steps. This contrasts with fully agentic systems where the next action is decided by the model at runtime. By enforcing determinism, the service guarantees predictable and repeatable executions. Recent research underscores the importance of this approach: using an explicit blueprint and a deterministic engine yields verifiable and auditable processes, as opposed to unpredictable agent behaviors [1]. In a similar vein, frameworks like LangGraph favor explicit state machines for multi-agent orchestration, resulting in workflows that are *“predictable and auditable”* – easier to debug and trust in enterprise settings [2].

- **Stateless Execution:** The service itself maintains no persistent state between steps. Each step in a workflow is executed independently, given its input and the overall flow definition, and it produces an output (and possibly a next-step decision). Any state needed (e.g. variables or results passed from one step to another) is included in the input/output data rather than stored in the server. This stateless design aligns well with serverless deployment and makes the system highly scalable. It also simplifies reasoning about execution, since each invocation of the service is pure (no hidden state from prior runs).

- **Workflow as Data (Blueprint):** Workflows are defined declaratively as data structures (a **blueprint** or **flow definition**) rather than hard-coded logic. An expert (developer) or an automated design tool specifies the sequence of actions, conditional branches, and allowed operations in a machine-readable format. The workflow blueprint can be represented in JSON or a similar structured schema, akin to how one might describe an API contract in OpenAPI. This blueprint is essentially a directed graph or state machine: each node represents a step (action to perform) and edges define transitions to the next step. The service's engine interprets the blueprint to decide what to do at each step and where to go next. Crucially, the LLM is *only* invoked within specific steps for bounded tasks – for example, to generate text or analyze data – but **never to choose the next step in the state machine** [1] . This ensures the flow's path remains under deterministic control of the blueprint.

- **Provenance and Auditability:** Every action taken by the workflow (LLM call, API call, decision branch, etc.) is logged and traceable. Because the sequence is predetermined (except for conditional logic which is still explicitly defined), we can provide a complete audit trail of what happened and why. This is critical in enterprise or safety-critical contexts where understanding the decision process is required. The structured nature of the flows makes it easier to inspect and verify correctness before execution, and to analyze outcomes after execution.

- **Limited LLM Scope (Safety):** The use of LLMs within the workflow is intentionally constrained. LLMs will be used for tasks like natural language processing, summarization, translation, or analysis – **not for making control decisions**. By bounding the LLM's role, we avoid the unpredictability that comes if an LLM "agent" had free rein. Essentially, the LLM behaves as a powerful subroutine under supervision. This mitigates risks of the LLM drifting off-task or producing unintended actions. It also allows insertion of validation steps (for example, after an LLM produces an output, the next step could validate that output against rules or schemas before proceeding).

- **Budgets and Resource Controls:** Each workflow (and even each actor or tool within the workflow) can be assigned a **budget** – for example, a limit on the number of API calls, the number of tokens processed by an LLM, or a time limit. These budgets are integrated into the flow logic. If a component exceeds its budget, the workflow can take a predefined path (such as aborting or invoking a fallback step). By designing budget constraints into the blueprint, we ensure the system cannot accidentally run away in loops or incur unbounded costs. In our scenario, for instance, the Persona service or LLM might only have a certain token budget; if translating or answering a question would exceed that, the flow would stop or switch to an error state. This built-in guardrail is another measure to keep executions safe and predictable.

- **Flexible Complexity:** The system is meant to handle simple to very complex workflows. A simple use-case might be a single-step call (e.g. "call this LLM with prompt X and return answer"). A complex use-case could involve multiple agents, branching logic, loops (controlled by explicit conditions), and tool integrations. The design should support both extremes. This implies the blueprint language must be expressive enough for conditions, branching (if/else), parallel execution (if needed), and looping (perhaps via recursive flows or explicit loop constructs) – all while remaining human-comprehensible. The service should execute any such defined flow **reliably**, since each part is under strict governance of the blueprint.

- **Integration with Knowledge Graphs (Future Scope):** Although not the first priority, the vision includes leveraging **semantic knowledge graphs** in conjunction with the flow engine. The idea is that the workflow steps and their context could be informed by a knowledge graph that contains relevant domain information or state. For example, a step might query a knowledge graph for facts that an LLM then uses in its prompt. Or the flow blueprint itself could be stored/ versioned as a graph of nodes (steps) and edges (transitions), enabling visualization and analysis of the workflow structure. By combining flows with semantic knowledge representations, we can achieve more powerful reasoning while still keeping the execution controlled. This is an area of ongoing research and will be explored as the service evolves.

Service Architecture and Components

The **LLM Workflows/Stateflow Service** architecture is composed of several key components that together enable the definition and execution of these workflows:

- **Workflow Definition (Blueprint):** At the core is the **workflow blueprint**, a complete specification of the states and transitions for a given process. This can be a JSON document or Python object (defined via dataclasses or Pydantic models) that lists all the steps in order, along with any branching logic. Each **Step** in the blueprint typically contains:
- An **action** identifier – e.g. `"call_LLM"` or `"call_service:persona.translate"` – which tells the engine what to do.
- **Parameters/input** for that action – e.g. the prompt to send to the LLM, or the data to send to an API. Parameters might include references to outputs of previous steps (for example, step 3 can use the result produced by step 2).
- **Result handling** info – e.g. where to store the result of this step (in a context object or variable map that gets passed along).
- **Transitions** – rules for what the next step is. In the simplest case, each step names a single next step. In more complex cases, a step may have multiple possible next steps depending on its result (for example, a branching step could say: *if evaluation score >= 8 go to step X, else go to step Y*). If a step is terminal, it can be marked as an **End** state.
- Optional **metadata** like timeouts or retries for that step, if an action might fail and should be retried.

The blueprint is analogous to a workflow script. The service includes a **Blueprint Interpreter** (or engine) which reads this definition and drives execution accordingly. Because the blueprint is declarative, we can inspect and validate it before running, and even visualize it as a flowchart.

- **Execution Engine:** The engine is a stateless component that takes a workflow blueprint plus the current state (inputs and context) and executes one step of the workflow. It can be implemented as a FastAPI endpoint (for example, `/execute_step`) which accepts a JSON payload containing the workflow definition (or a reference to it), the current step to execute, and any necessary input data or context. The engine performs the action of that step and returns the result (and the identifier of the next step to execute, if any). Important responsibilities of the engine include:
- **Action Dispatch:** Based on the step's action identifier, call the appropriate function or external service. For example, if the action is `call_LLM`, the engine will invoke the configured LLM API (such as an OpenAI or Anthropic model) with the given prompt and parameters. If the action is `persona.translate` or `persona.respond`, the engine calls the Persona service endpoint. If it's a generic HTTP API call, the engine will make that HTTP request (possibly the blueprint could contain the URL or a reference to a pre-registered service with credentials).
- **Budget Checking:** Before executing an action, the engine checks the remaining budget for that actor/service. This could be implemented by maintaining a transient budget counter in the context that tracks tokens or calls. For instance, if the Persona service has 100 tokens budget and translating the message is estimated to use 30, the engine verifies budget >= 30, deducts it, then proceeds. If budget would be exceeded, the engine can skip to a special termination or error step defined in the blueprint.

- **Result Handling:** After performing the action, the engine takes the output and places it in the workflow's state (for example, storing it under a variable name). If the blueprint specifies post-processing (like simple transformations) or validations (e.g. ensure the result fits an expected format), those are done here as well.
- **Next Step Logic:** The engine determines which state to execute next. In straightforward sequences, the blueprint might indicate a `next` step explicitly. If the current step was a branching/choice step or if the action outcome dictates the path, the engine evaluates the condition and picks the appropriate next step. For example, an evaluator step might branch to different follow-up steps depending on the score (high score leads to a "successful completion" branch, low score leads to an "escalation" branch). If there is no next step (the step was marked as an end), the workflow is complete and the engine returns the final result of the workflow.

Notably, the engine is **agnostic to the overall workflow** – it doesn't keep a running memory of all previous steps beyond what is carried in the input context. This makes it easy to scale or even to *pause and resume* workflows by simply not calling the next step immediately. It is the responsibility of the **Orchestrator** (described next) to drive the engine through the steps.

- **Workflow Orchestrator:** Since the service executes one step at a time, we need an orchestrator to manage multi-step workflows from start to finish. This could be a separate client application, a calling service, or even a simple loop in a script that keeps invoking the engine until the workflow ends. For example, suppose we have a workflow with steps A -> B -> C. The orchestrator would:
- Call `/execute_step` for step A with initial input. Receive result and next step = B.
- Call `/execute_step` for step B with result of A. Receive result and next step = C.
- Call `/execute_step` for step C with result of B. Receive result and next step = none (end).
- Collect final result.

The orchestrator could be implemented as part of the client using this service, or we could provide a utility in the service that takes a whole blueprint and automatically steps through it. However, having the orchestrator outside the core engine adds flexibility: workflows can be paused, inspected mid-run, or even modified between steps if needed (for advanced use cases).

In a serverless deployment (e.g., AWS Lambda via OSBot-FastAPI-Serverless), the orchestrator might be a state machine service (like AWS Step Functions itself or Azure Durable Functions) that triggers the next lambda invocation. Alternatively, a simple loop in an API endpoint (if using a container or persistent service) could also orchestrate the steps synchronously.

- **Integrated Services and Tools:** The workflow steps can involve various external services:
- An **LLM Service**: e.g., an API call to an OpenAI or a local model. The flow can specify which model and prompt to use. The LLM service endpoint would need to be reachable by the workflow engine and secured (API keys, etc.). The service may have multiple models (GPT-4, GPT-3.5, etc.) and the blueprint could choose which one by an identifier.
- The **Persona Service**: (at `persona.qa.mgraph.ai` as mentioned) which has two main capabilities:
    - *Persona Translation*: take an input message and translate or rephrase it for a target persona or in a target language (e.g., explain a technical incident in board-level terms, or convert English to Portuguese, etc.).
    - *Persona Response*: given a prompt or question, respond **as** the persona (generating an answer the persona would give). This service is itself likely powered by an LLM under the hood, but from our workflow's perspective it's a black-box API that we call with certain parameters. The workflow can include steps that call `persona.translate` or `persona.respond` with the appropriate persona and message.

- **Evaluation Service**: an AI or rule-based system that evaluates a given piece of content. In the example scenario, this is used to rate how well a response was understood or how effective it was. This could be another LLM prompt (e.g., asking an evaluator model to score the answer on clarity), or a more deterministic function that checks for certain keywords, etc. We treat it as an external call (perhaps another endpoint like `evaluator.rate_response`).
- **Other APIs/Tools**: The system could integrate with any HTTP API or internal function. For instance, steps could call a database, send an email, trigger a CI/CD pipeline, run a shell command (if allowed), etc. These would be included by defining new action types in the blueprint and teaching the engine how to execute them. The design is extensible; new action handlers can be added to the engine plugin-style.
- **Budget Manager:** As mentioned, budgets for each actor/service must be enforced. A simple implementation is to include budget counters in the workflow context passed through steps. For example, the context could have `{ "budget": {"persona": 1000, "llm": 1500, "evaluator": 500} }` indicating remaining token or API call counts. Each time the engine is about to call one of these, it subtracts an estimated cost. If the cost is not known exactly (e.g., tokens to be consumed by an LLM call depends on output length), the engine can either make a conservative guess or get the info after the call (some LLM APIs return token usage). If a budget hits zero or goes negative, the engine can trigger a special transition (for example, go to a "Budget Exceeded" fail state). The blueprint can define what to do in that case (maybe notify the user, or just terminate). The **Budget Manager** could be just a piece of logic in the engine or a separate helper module.
- **Logging & Monitoring:** Every request to the engine and every action invocation should be logged. This includes inputs, outputs, and any decision points. For debugging complex workflows, it is invaluable to see a trace of steps. We can integrate this with the OSBot-FastAPI's event system (which already provides request/response tracking and can store events) [3]. Logs can be written to a database or a cloud logging service. We might also have a **monitoring dashboard** that shows active workflows, their progress, and any errors. For long-running workflows (if any), monitoring would show the current state and waiting transitions. Given the stateless nature, most workflows will run quickly or be orchestrated externally, but monitoring is still useful for analyzing results and performance. We will also capture metrics like time per step, tokens used, etc., to help optimize the flows and LLM usage.
- **Security & Access Control:** As an OWASP Security Bot project, security is important. The service should authenticate and authorize requests (using API keys or tokens, as supported by OSBot-FastAPI middleware [4]). Only authorized systems or users should be able to submit a workflow for execution or call the engine. Also, the actions allowed in a workflow might be gated: for example, we might restrict certain flows from calling arbitrary URLs unless explicitly allowed, to prevent misuse. Each integrated service (Persona, LLM, etc.) will have its own credentials (API keys), which the engine must handle securely (likely stored in environment or a vault, not hard-coded in the blueprint). The blueprint might reference a service by name and the engine knows which credentials to apply.

In summary, the architecture separates the *declaration* of what to do (blueprint) from the *execution* of how it's done (engine and orchestrator), with clear interfaces to external AI services and strict control logic enveloping any LLM calls. This design maximizes reliability and clarity, ensuring that even as we automate complex tasks with AI, the process remains transparent and governable.

Workflow Definition and Standards

Defining the workflow blueprint in a robust way is a critical aspect of this service. We want a format that is **expressive, standard-compliant (when possible), and easy for developers to author and maintain.** Given that our implementation language is Python (with **OSBot-FastAPI** and **OSBot-FastAPI-Serverless** frameworks), we have a couple of choices for how to represent workflows:

1. **Python Type-Safe Classes (Code as Blueprint):** We can create Python classes (using OSBot's Type_Safe or Pydantic models) to represent the elements of a workflow – e.g., a `Flow` class containing a list of `Step` objects, where each Step has fields like `id`, `action`, `parameters`, `transitions`, etc. Developers can then construct these classes in code or load them from a JSON. OSBot-FastAPI will automatically handle conversion between these classes and JSON schemas [3]. This means we get strong typing and validation for free. For example, if a step is missing a required field or has an invalid next step reference, the model validation can catch it before execution. This approach treats the blueprint almost like writing a program (in Python), which is then serialized to JSON when needed.

2. **JSON/YAML Workflow Schema (Data as Blueprint):** Alternatively, we define a pure JSON (or YAML) schema for the workflow – a text-based format that could be authored by hand or generated by tools. There are **existing standards and schemas** in the industry that we can draw inspiration from:

3. **BPMN 2.0:** A long-standing standard for business process modeling (usually visual diagrams stored in XML) [5]. BPMN is very expressive (supports events, gateways, etc.), but it might be overly complex for our needs and not JSON-friendly by default.
4. **BPEL:** An older XML-based language for web service orchestration [5]. Also quite heavy and tied to WS-* services.
5. **AWS Step Functions (Amazon States Language):** A JSON-based state machine definition used in AWS Step Functions [6]. This is a practical and widely used format. It represents workflows as a set of states (Task, Choice, Parallel, etc.) with a `StartAt` and explicit `Next` transitions or `Choice` branches. It's quite suitable for our concept since it inherently models step-by-step execution with choices. The downside is that it's AWS-specific in some of its integration details, but we could adopt the structure. For example, we'd have "Task" states for calling LLM or Persona, and "Choice" states for branching on evaluation results, etc., all expressed in JSON.
6. **Azure Logic Apps:** Similar to Step Functions, uses JSON (and often designed via a visual editor) [7]. Also could be a reference, though tied to Azure connectors.
7. **Workflow Description Languages (WDL/CWL):** These are specialized languages for scientific and data workflows [8]. They emphasize reproducibility and usually model batch processing of data (like DNA sequencing pipelines). They might be too domain-specific for our interactive use-cases, but they show how to define steps, inputs, and outputs clearly.
8. **Argo Workflows / Tekton:** Kubernetes-native workflow engines using YAML [9]. They often focus on CI/CD pipelines and container tasks. The concept of defining DAGs of steps is similar, though Argo's YAML could be more low-level (each step is basically a container spec).
9. **Custom DSLs:** Many teams end up creating their own lightweight JSON/YAML schemas for workflows [10]. This is likely the approach we will take: design a JSON schema tailored to our AI workflow needs, while borrowing ideas from the above standards for structure and best practices. For instance, we might incorporate Step Function's idea of states and transitions, but simplify it, or include a field for natural language description like BPMN does, etc.

Given that we aim for a **type-safe** and easily maintainable solution, our plan is to define a **JSON-based schema** for the workflow and also represent it with Python classes for convenience. We can start from scratch or use a nascent standard like **FlowSpec**. *FlowSpec* is an open initiative to create a standardized JSON schema for AI automation workflows [11]. It recognizes that many workflow tools share a flow-chart backbone, and it attempts to unify this in a portable way. In FlowSpec, a workflow is defined by a title, description, a list of steps, and transitions between steps [11]. Each step has fields for what action to execute, its inputs, expected outputs, and what the next step(s) are depending on outcomes. It even allows global default transitions (like what to do on any failure) [12] [13]. FlowSpec also enumerates existing workflow standards (as we did above) to validate the approach of a common schema [13].

After researching these options, **our recommendation** is to adopt a JSON state machine schema inspired by AWS Step Functions and FlowSpec. This gives us a known structure (states, next, choice, etc.) but we will customize it for our needs (for example, integrate the notion of budget and our specific action types). We will keep the schema human-readable and not too verbose. For instance, a simple flow with two steps might look like:

```
{
  "workflowName": "Simple Q&A",
  "startAt": "AskQuestion",
  "states": {
    "AskQuestion": {
       "type": "Task",
       "action": "call_LLM",
       "parameters": {
          "prompt": "Answer the user's question: {user_question}"
       },
       "resultVar": "answer",
       "next": "EvaluateAnswer"
    },
    "EvaluateAnswer": {
       "type": "Task",
       "action": "evaluator.rate_response",
       "parameters": {
          "response": "{answer}"
       },
       "resultVar": "score",
       "end": true
    }
  }
}
```

In this pseudo-JSON: - `startAt` specifies the entry step. - We have two states: one calls an LLM to get an answer, the next calls an evaluator to score that answer, then ends. - We use placeholders like `{user_question}` and `{answer}` to indicate passing data between steps (the engine would replace those at runtime with actual values from context). - This format is quite similar to Amazon States Language (each state has a `Type` and either a `Next` or `End`) [14] [15], but with an `action` field that is our custom addition to specify what the Task does (since we are not tying directly into AWS Lambda ARNs as AWS does [16]).

We will formalize such a schema and provide a JSON Schema definition for it (so it can be validated). The use of Python with OSBot-FastAPI means we can also create corresponding classes. For example, a `TaskState` class and a `ChoiceState` class that inherit from a base `State` class, etc., enabling developers to construct workflows in Python fluidly. The **OSBot-Fast-API** toolkit will assist by ensuring these classes convert to Pydantic models easily, preserving the strong types [3]. This approach satisfies our need for a clear contract for workflows, while leveraging existing best practices from industry standards [13] [17].

To summarize, the workflow definition will likely be expressed as JSON but with first-class support in Python. It will incorporate ideas from state machine standards (like having explicit states, transitions, start/end, etc.) and will be designed to be **readable, easy to modify, and rigorous**. By doing this, we make it easier for developers to create new workflows or adjust existing ones, and possibly even enable *LLM-assisted workflow authoring* in the future – for instance, an LLM could take a high-level description and output a draft JSON workflow, which a developer then reviews and fine-tunes. The use of a standard schema also opens the door to visualization tools or workflow editors down the line.

Example Workflow: Persona-Based Communication and Evaluation

To illustrate how the stateflow service works, let's walk through a detailed example workflow. This scenario involves translating and conveying a critical message between two types of personas in an organization, and evaluating the communication's effectiveness. We will use the previously described actors: - **Actor A:** The originator of the message (could be a human user or an automated alert). In our scenario, the message is: *"A ransomware attack has hit Division X, which will impact the P&L (profit and loss) for this quarter."* - **Actor B:** The Persona Service, which can assume different personas. We will use it in two modes: - *Translator mode:* to rephrase a message for a target persona's understanding. - *Responder mode:* to generate a reply as if coming from a persona. - **Actor C:** The Evaluator service, which will judge the quality of responses (e.g., does the response answer the question clearly, does the target persona understand the message, etc.).

The organizational context is that **Board Members** care about financial terms like P&L but might not understand technical cybersecurity jargon, whereas **CISOs (Chief Information Security Officers)** understand ransomware but might not grasp business impact jargon. Our message contains both technical (ransomware) and financial (P&L) terms, so it's challenging for either persona to fully understand without translation.

We will construct a workflow that explores different communication paths:

1. Direct Communication to CISO (No Translation):
Actor A sends the original message directly to a CISO persona (via Actor B's responder mode acting as a CISO). The flow steps might be: - *Step 1:* `persona.respond` as **CISO** with input = "Ransomware attack on Division X will impact P&L this quarter."
→ (Actor B generates a response as it thinks a CISO would reply. This CISO likely understands the ransomware part but may be confused or less concerned about P&L specifics. The response might say something focusing on cybersecurity mitigation but not address financial impact fully.) - *Step 2:* `evaluator.rate_response` on the CISO's reply, with criteria like *completeness*, *clarity*, *appropriateness for the question*.
→ (Actor C returns a score or feedback. We expect this might be a mediocre score if the CISO persona missed the financial aspect.) - *Step 3:* End. (We record the score and perhaps the content of the CISO's answer.)

Expected outcome: The CISO's answer might mention technical steps (e.g., *"We are investigating the ransomware attack on Division X and working to contain it."*) but not translate that into business terms.

The evaluator might note that the board (who cares about P&L) would not get a full picture from this answer. The score could be low or moderate.

2. Direct Communication to Board Member (No Translation):
Actor A sends the same message directly to a Board Member persona (Actor B acting as a board member): - *Step 1:* `persona.respond` as **BoardMember** with input = "Ransomware attack on Division X will impact P&L this quarter."
→ (Actor B generates a response a board member might give. The board member persona might latch onto the P&L impact but be unsure about the technical details, possibly responding with something like *"How severe is the ransomware attack and what are the projected losses?"*) - *Step 2:* `evaluator.rate_response` on the Board Member's reply.
→ (We expect the board member's answer might not be directly useful because the board persona might actually ask questions or express confusion about the ransomware aspect. The evaluator likely scores this low in terms of addressing the problem, since the board member persona didn't provide a solution or clear action.) - *End.*

This path shows how a mismatched communication (technical message to non-technical persona) might fail. The board member didn't provide a satisfying answer because they themselves didn't fully understand the technical side. The evaluator would likely flag that the communication was ineffective.

3. Translated Communication to CISO:
Now we improve the communication. Actor A's original message will first be translated to the CISO's "language" (i.e., reframed in cybersecurity terms), then delivered to the CISO persona, and evaluated: - *Step 1:* `persona.translate` target=**CISO**, input = "Ransomware attack on Division X will impact P&L..."
→ (Actor B returns a **translated message** that a CISO would immediately grasp. For instance, it might elaborate the technical threat and downplay financial jargon: *"Division X has been hit by ransomware, affecting operations; this could have a significant business impact this quarter."*) - *Step 2:* `persona.respond` as **CISO** with input = **translated message from Step 1**.
→ (Now, receiving a message phrased in his context, the CISO persona can respond more appropriately. The answer might be like: *"Understood. We have isolated the affected systems and are initiating incident response. We estimate recovery in 48 hours. Financial impact is being assessed in collaboration with finance."* This is a more complete answer covering both tech and acknowledging financial impact, because the question was framed in terms the CISO cares about.) - *Step 3:* `evaluator.rate_response` on the CISO's new reply.
→ (Actor C would likely give a higher score here, since the response is clear, addresses the issue, and bridges to business impact. The evaluator might note that the communication was effective for the target audience.) - *End.*

We expect this translated workflow to yield a good outcome: the CISO persona understood the question after translation and responded in a way that likely satisfies a board or oversight evaluator.

4. Translated Communication to Board Member:
Similarly, translate the message for a Board Member, then get a response: - *Step 1:* `persona.translate` target=**BoardMember**, input = original message.
→ (This might produce something like: *"We estimate a hit to this quarter's profits due to a cyber incident (ransomware in Division X)."* Essentially explaining ransomware impact in terms a board cares about, possibly avoiding jargon.) - *Step 2:* `persona.respond` as **BoardMember** with input = translated message.
→ (Now the board persona, fully aware of the financial framing, might respond appropriately, e.g.:

"Understood. Ensure all necessary resources are allocated to IT to resolve this quickly. Let's prepare a statement for stakeholders about the financial impact.") - *Step 3:* `evaluator.rate_response` on this reply.
→ (Likely another high score – the board member persona's answer is on point when the question was phrased in their terms.) - *End.*

This shows that with proper translation, even a non-technical persona can engage effectively.

5. Back-and-Forth Dialogue (CISO ⟷ Board, Mediated by Translations):
We can extend the scenario to simulate an interactive dialogue between the CISO and Board Member personas. The idea is to have multiple turns: - First, the CISO receives a translated question (as in #3) and responds as CISO. - Then take the CISO's response, translate it for the Board, get a Board persona reply. - Then translate that reply back to CISO's terms, get CISO's next response. - Continue this exchange for a few iterations or until a **budget limit** is reached (to prevent infinite loops).

In the workflow blueprint, this could be represented by a loop or recursive transitions. For example: - Step 1: `persona.translate` to CISO (original message) -> output ciso_msg. - Step 2: `persona.respond` as CISO (ciso_msg) -> output ciso_reply. - Step 3: `persona.translate` to Board (ciso_reply) -> output board_msg. - Step 4: `persona.respond` as Board (board_msg) -> output board_reply. - Step 5: **Loop condition**: If `board_reply` or some context indicates conversation should continue AND budgets remain, go back to Step 1 (or a specific step) with `board_reply` now serving as the "original message" (Actor A's input) for the next round, targeting CISO again. - If loop ends (either a set number of rounds reached or budget exhausted), proceed to evaluation or finalization: - Step 6: `evaluator.rate_response` on the final response or on the overall dialogue quality. - End.

This looping construct is explicitly controlled. The blueprint would contain a **Choice** or condition check after Step 4 to decide whether to loop or exit. The *budget* for each persona ensures that, say, we don't allow more than N exchanges or Y tokens. For instance, we might give each persona service 3 calls budget. Each `persona.respond` call uses 1. So at most 3 rounds of responses per persona can happen (which is 3 CISO replies and 3 Board replies, for a total of 3 cycles) before the budget prevents further calls.

During this back-and-forth, each translation ensures both parties understand each other's messages in their own context. The **Evaluator** at the end might evaluate the overall success of the communication. Perhaps it looks at the final outcome: did they reach a mutual understanding or plan? We could even have the evaluator step after each reply, storing intermediate scores, but in practice it might suffice to evaluate at the end or only log the conversation.

This complex example demonstrates the power of the workflow approach: - We can coordinate multiple AI calls (translations, persona responses, evaluations) in a sequence that achieves a larger goal (effective communication). - Because it's all in a defined flow, we avoid chaos: e.g., the Board and CISO personas will not talk over each other or go off on tangents; they only respond when prompted by the workflow. - If something fails (say one of the steps returns an error or empty response), we could have failure paths defined. For example, if `persona.respond` fails due to no available LLM, the blueprint could go to a step that sends a default apology message or logs the failure. - The budget prevents infinite loops or runaway costs, which is something ad-hoc agent loops might suffer from.

In summary, this Persona Communication workflow shows a realistic use-case where **deterministic orchestration of LLM-powered services** adds significant value. It ensures that two different knowledge domains (technical vs business) can interact via AI intermediaries in a structured manner.

The *stateflow service* makes it feasible to design such an interaction as a series of controlled steps, rather than leaving the entire conversation flow to an unpredictable AI agent. Each step's outcome is evaluated and can trigger specific next steps, which is exactly the kind of fine-grained control we need for enterprise applications.

Additional Workflow Examples

Beyond the persona translation scenario, the LLM Workflows service can support a wide range of other workflows. Here are a few **example use-cases** to demonstrate its versatility:

- **Simple LLM Q&A Workflow:** *("Single-step answer")* – The user provides a query, and the workflow simply calls an LLM to get an answer and returns it. This is essentially a one-step workflow (plus maybe an evaluator or format step). While trivial, it shows how even a simple LLM invocation can be wrapped in the workflow for consistency (logging, budgeting, etc.). For instance, a question-answering bot could be just a workflow with one Task: `call_LLM` (with a certain prompt structure) and then End.

- **Knowledge Base Retrieval and Answering:** *("Tool-augmented query")* – A workflow can integrate a search or database lookup before calling the LLM. Steps might be: (1) take a user question, (2) use a custom action `call_web_search` or `query_knowledge_graph` to retrieve relevant info, (3) feed the results into an `call_LLM` step that formulates an answer using those results, (4) maybe an evaluator step to check confidence or filter out any disallowed content. This deterministic sequence ensures the LLM's answer is grounded in retrieved data (addressing factuality), and each part is controlled (for example, if the search returns nothing, we could have a conditional branch to skip the LLM call and respond with "no data found").

- **Automated Code Assistant Workflow:** – Consider a developer asking for code assistance. The workflow could involve multiple specialized steps: (1) `call_LLM` with a "planner" prompt that breaks down the request (e.g., "write a function to do X") into tasks, (2) loop through sub-tasks where for each task we call either a coding LLM to generate code or a testing tool to verify the code, (3) integrate results, (4) evaluate final code. For example, the first LLM might produce a pseudocode or list of steps, the workflow then calls a code-generation model to implement each step, then calls a compilation or test action to check it, if a test fails perhaps branch to a debugging LLM step, etc. Using a workflow ensures each step (planning, coding, testing, fixing) is done in order and under budget. This is much safer than an autonomous coding agent that might go into an infinite loop or execute code unsafely – our workflow can explicitly restrict what happens (like only allow running tests in a sandbox, etc.).

- **Content Moderation Pipeline:** – An enterprise might use a workflow to filter and respond to user-generated content. For example: (1) `moderation_model` step (could be an LLM or a dedicated model) to classify a piece of text (is it hate speech, spam, etc.?), (2) a `choice` state that branches: if content is OK, proceed to next step, if not OK, go to a rejection message step or escalation, (3) maybe an `auto-response` step that uses an LLM to draft a polite reply or explanation, (4) an `approve` step where either automatically send the reply or require a human approval (this could be an integration point where the workflow pauses until a human intervenes – possible by having the orchestrator not call the next step until a signal). This kind of workflow could automate moderation while keeping humans in the loop for tough cases, all defined by policy in the blueprint.

- **Incident Response Workflow:** – In a cybersecurity context (relevant to OWASP), imagine a workflow triggered by a security alert. Steps could be: (1) parse the alert details (maybe using regex or an LLM to summarize), (2) `choice` to categorize severity, (3) if severe, call a script or API to isolate affected systems, (4) call LLM to draft a notification email to the IT team or management, (5) log the incident to a database. Each of these is a deterministic step. The LLM is used in a constrained way (only to generate the email text), while decisions like "if severe then isolate systems" are hard-coded in the flow logic (not left to the AI). This ensures that important actions (like isolating systems) happen exactly when they should according to a predefined protocol, but we still benefit from AI in parsing and communicating information.

- **Multi-Language Customer Support:** – A customer writes in with a query in language X. The workflow: (1) detect language (maybe a small model or library call), (2) if not English, `translate` to English via an LLM or translation API, (3) use an LLM to draft an answer in English, referencing a support knowledge base if needed, (4) translate the answer back to the customer's language, (5) send the reply via an API. This workflow uses two LLM calls (one to answer, and possibly the same or another to translate) and ensures the final answer is in the customer's language. By orchestrating it, we can ensure translation happens both ways and include fallback steps (if translation fails, perhaps route to a human agent). It's a controlled agent that can autonomously handle many support tickets in multiple languages without ever deviating from the defined process.

These examples scratch the surface. Essentially, any time we want an **LLM or AI-driven process with multiple steps** and we care about controlling those steps, this service can help. It provides the skeleton to plug in various AI and non-AI functions into a flowchart of actions.

By keeping the workflows **declarative** and using this service, organizations can codify complex procedures that involve AI into a form that's *transparent, testable, and tunable*. Need to change the persona or the prompt? Just update the blueprint. Want to add a step to log to a new database? Add it to the blueprint. Because the execution is isolated per step, these modifications won't affect other steps' correctness. This modularity and clarity is much harder to achieve if one tries to hard-code logic intermingled with LLM prompts in a single blob. Our service enforces good separation of concerns.

Implementation Considerations (Python & OSBot Framework)

The service will be implemented in **Python 3.11+** (per OSBot-Fast-API requirements [18]) using the **OSBot-Fast-API** library and its serverless extension. Here we outline how we leverage these technologies and other implementation details:

- **OSBot-Fast-API for Type Safety:** OSBot-Fast-API provides a strong type-safe layer on top of FastAPI [3]. We will define the data models for our workflow blueprint (and related requests/ responses) as classes using OSBot's `Type_Safe` or Pydantic. This ensures that when a workflow JSON is received, it's automatically validated against our schema. It also makes it easy to return structured responses. For instance, the `/execute_step` endpoint can be defined to accept a `WorkflowStepExecutionRequest` object (containing the blueprint or reference and current step data) and return a `WorkflowStepResult` object. OSBot-Fast-API will handle converting those to JSON for us, and we can be confident in the structure. Strong typing will catch mistakes early – e.g., if a transition refers to a step that doesn't exist, we can detect that when loading the workflow.

- **Serverless Deployment:** OSBot-Fast-API-Serverless enables deploying the FastAPI app to AWS Lambda easily. Our service, being stateless and lightweight in memory (each step execution is quick), is a good candidate for serverless. We could deploy the entire service as a Lambda behind an API Gateway. Each step execution call would be a separate invocation (which is fine given statelessness). The benefits: scaling automatically with load and zero server maintenance. We do need to be mindful of cold start times (Python Lambdas can have a few seconds cold start; using smaller models or warming mechanisms might be considered if needed). Also, token-based LLM calls can be slow (hundreds of milliseconds to seconds), but those are external API waits; the Lambda timeout should be set sufficiently high to allow an LLM call to complete (perhaps 30 seconds or more for large prompts). For now, we focus on functionality, and we have the flexibility to also run the service in a container or on a VM if needed (FastAPI is versatile).

- **Integration of LLM APIs:** We will integrate with LLM providers via their Python SDKs or HTTP APIs. Likely, OpenAI's API (for GPT-4 or others) will be used initially (assuming we have keys). Calls to these will happen inside the engine's action dispatch. We must handle errors (network issues, rate limits) gracefully – possibly by catching exceptions and either retrying (with backoff) or moving to a failure state. We should also use streaming responses only if necessary; otherwise synchronous calls returning the full output are simpler. For local LLMs or alternative providers, we can abstract the LLM call behind a common interface so that switching out is easy (for example, have a `LLMService` class with a method `generate(model, prompt)` that can call OpenAI, or HuggingFace pipeline, etc., based on configuration).

- **Testing Workflows:** Because of the deterministic nature, we can unit test workflows by simulating the orchestrator. For a given blueprint, we can run the engine step by step and assert the final outcome or intermediate states. We can also create dummy action handlers for tests (e.g., instead of calling the real LLM API, use a stub that returns a fixed string or uses a local small model). This way, we can verify logic (like branching) without external dependencies. OSBot-Fast-API's testing utilities (like the built-in test server [4]) will help in writing these tests. Each workflow example can have an automated test case that ensures it runs to completion and yields expected evaluator scores, etc., which is important for continuous integration.

- **Performance and Caching:** Calling LLMs is the slowest part. We might implement caching at the step level. For example, if the same `persona.translate` is called with the exact same input frequently, we could cache the result to avoid redundant API calls. This could be done in-memory (for a single lambda invocation sequence) or even persisted (like a small Redis or DynamoDB cache keyed by input). However, caching needs to be designed carefully (e.g., LLM outputs might not be identical every time unless using deterministic prompts). Perhaps more straightforward is to avoid duplicate calls in the same workflow execution – since a single workflow might reuse a result anyway via variables.

- **Choosing Standards Libraries:** For state machine logic within Python, we might use or draw inspiration from libraries like `transitions` (a Python state machine library) or others, but given our custom requirements, we will likely implement the transition handling ourselves. The logic is not too complex given a proper data structure for the blueprint.

- **Error Handling and Recovery:** If a step errors out (throws an exception, or returns a response indicating failure), the engine should catch that and either:

- Move to a predefined error state if the blueprint defined one (like how AWS Step Functions has a `Catch` mechanism).

- Or return an error back to the orchestrator. Since orchestrator is external, it's probably better to handle it within the workflow. We can allow steps to have a `on_error: <stepId>` field. This way, e.g., if an LLM call fails, we go to a specific step (maybe an apology message, or a cleanup). If no on_error is specified, the engine can return an error code and the whole workflow aborts. This aspect should be defined in our schema for completeness.
- Logging the error is important regardless, for debugging.

- **Collaboration and Iteration:** As development proceeds, the team (human developers) will refine the blueprint schema and engine logic, often with the assistance of LLMs for ideas or troubleshooting. This synergy will continue as we implement new features. For instance, if we want to introduce a new kind of step (say, a Parallel step to do two things concurrently), we might consult resources or have an LLM suggest how to implement thread pools or async patterns in FastAPI. However, all changes will be reviewed and tested by developers to ensure they meet the determinism and security criteria.

- **Documentation and Accessibility:** We will document the schema and usage of the service thoroughly (potentially even auto-generating part of the docs from the schema, similar to how OpenAPI does for APIs). Since Dinis Cruz and the team are building this in the open (likely on GitHub), the documentation will credit the contributions of both the developers and the AI (ChatGPT) that helped along the way. This technical brief itself can serve as a living document to guide implementation, and as we integrate feedback and real-world testing, the design may be adjusted. The flexibility of our approach (thanks to Python and JSON) means we can iterate quickly.

In conclusion, the **LLM Workflows/Stateflow Service** is a cutting-edge approach to making LLM-based systems more robust, transparent, and controllable. By blending established workflow orchestration concepts with the latest AI capabilities, and implementing it with modern Python frameworks, we aim to create a service that developers and AI systems can **collaboratively use and improve**. It will empower the creation of AI-driven applications that have the creativity of LLMs *and* the reliability of traditional software – a combination that is increasingly essential in high-stakes applications [19] [20]. With this foundation, we anticipate a new class of solutions where humans specify the *roadmap* (workflow) and AI fills in the *details* (content), all under a structure that ensures safety and effectiveness.

---

[1] [19] [20] Blueprint First, Model Second: A Framework for Deterministic LLM Workflow
https://arxiv.org/html/2508.02721v1

[2] LangGraph vs AutoGen: How are These LLM Workflow Orchestration Platforms Different? - ZenML Blog
https://www.zenml.io/blog/langgraph-vs-autogen

[3] [4] [18] osbot-fast-api · PyPI
https://pypi.org/project/osbot-fast-api/

[5] [6] [7] [8] [9] [10] [11] [12] [13] [17] GitHub - woodyhayday/FlowSpec: FlowSpec: Automation Workflow Schema - A lightweight JSON schema for defining automations and multi-step workflows. Designed for AI Automation Workflows
https://github.com/woodyhayday/FlowSpec

[14] [15] [16] Amazon States Language
https://states-language.net/