

Adaptive Narrative Multi-Agent Engine: Architectures and Best Practices

We compare and synthesize leading open-source **multi-agent/NPC simulation frameworks**, memory architectures, narrative-director designs, and API-routing strategies to guide building an adaptive narrative engine. Our focus is on autonomous background simulation, intermittent player interrogation, and dynamic event injection, with tiered LLM usage (including English and Persian support via OpenRouter).

1. State-of-the-Art Frameworks & Agent Runtimes

Modern multi-agent frameworks combine LLM-powered agents with memory, planning, and communication modules. For example, **Stanford's Generative Agents** architecture extends LLMs with *experience logs*, *memory synthesis*, and *retrieval* to simulate human behavior. Agents record all experiences in natural language, generate high-level reflections, and retrieve relevant memories to plan actions [65†L29-L37]. Smallville (a Java/JavaScript port of Generative Agents) runs a headless simulation server where each tick *observes*, *decides*, and *acts* for every agent. The engine saves world state each step and updates each agent's memory and actions when `updateState()` is called (also allowing mid-tick "interview" interruptions via its HTTP API [20†L239-L247] [36†L7-L14]).

ElizaOS (TypeScript) provides an "agentic OS" with a built-in `AgentRuntime` message loop. Its core defines agent primitives (memory, state, messaging) and supports plugins [68†L1-L4]. Agents can run in parallel and broadcast messages via built-in channels or a pub/sub system. Eliza agents can be interrupted by human input (e.g. chat prompts) at any time, with the runtime handling asynchronous tasks and state changes.

Letta (MemGPT) focuses on *structured memory tiers*: a short-term message buffer, editable "core memory blocks", long-term recall logs, and archival memory in external stores [70†L52-L60] [70†L123-L130]. Letta's APIs manage shifting info between these tiers (via LLM summarization, external DB lookups, etc.) so agents maintain coherence over time.

LangGraph introduces a *graph-based orchestrator* for multi-agent workflows. Each agent is a node, and edges route messages or tasks between them. Control flow is managed by graph edges – essentially modeling agents as states in a directed graph [27†L68-L73]. This supports complex pipelines (e.g. supervisor/sub-agent hierarchies) and can be adapted to game AI by treating NPCs as graph nodes that pass tokens or events along edges.

CAMEL-AI (Python) is a research framework for large-scale multi-agent simulation. It emphasizes *scalability* and *stateful memory*, allowing up to millions of agents with persistent memory [67†L279-L287]. CAMEL provides ChatAgent and Planner components and tracks each agent's memory and communication. (However, CAMEL is geared toward research benchmarks rather than interactive games.)

SkyAGI (Python) implements a playable demo of generative agents in a dialogue adventure. It uses LLMs to drive NPC conversation and actions with memory, producing surprisingly coherent emergent behavior. It follows the Generative Agents idea but adds a simple “game loop” CLI: on each turn, every character can converse or move, automatically updating a shared environment state each tick.

SillyTavern is not an agent framework but a rich **UI frontend** for character chat. It uses “character cards” (prompt templates) and supports multiple LLM backends (OpenAI, KoboldAI, etc.) [35†L23-L32] [35†L78-L86]. Its architecture branches into release vs staging builds, but fundamentally it just sends chat to one AI at a time – it does not simulate background agents or memory. SillyTavern is an example of a user-facing chat interface, not a simulation engine.

Execution Loops: In practice, most frameworks use a *tick loop* or *event loop*. Generative Agents (Stanford/Smallville) use a discrete tick scheduler: each step, agents *observe* the (possibly changed) world state, then *plan* (LLM call), then *act* (output text or movement) before saving state. ElizaOS agents run in a loop listening for messages and can be interrupted via callbacks. Letta’s agents maintain a continuous chat thread (new player questions can be injected anytime, updating memory afterward). LangGraph’s agents execute on DAG-driven triggers rather than real time. All systems persist state via JSON or databases between ticks. Inter-agent broadcasting is typically done by having agents write to a shared world state (like Smallville’s environment JSON or CAMEL’s message queues) so all others “observe” those events on their next step.

Best Base Engine: For an open-source back end, a Python framework (FastAPI or similar) is recommended, given rich LLM integrations (LangChain, etc.) and vector DB support. For example, FastAPI can serve agent state APIs and queue tick updates. A minimal template might be a lightweight orchestrator that runs a Python tick loop (triggered by a scheduler or manual call), calling a handler for each agent and saving state (as in the Generative Agents code [19†L1962-L1970]). If a Node/TypeScript solution is preferred, ElizaOS provides a comprehensive example with HTTP APIs and a plugin system. However, Python’s ecosystem currently has more libraries for memory/RAG processing.

2. Context Window & Dynamic Memory Architecture

Memory Streams: Agents need multi-tier memory. Implement a **short-term buffer** (sliding window of most recent dialogue/actions) directly in the prompt. Maintain **vector-indexed memories** in an external database (PostgreSQL + pgvector, Qdrant, or similar). Each time an agent acts or receives input, embed the new information and append it to the memory DB. At query time, retrieve the top-k relevant memories by embedding similarity and inject those into the prompt (retrieval-augmented generation) [46†L340-L349]. For example, split each saved memory into chunks, embed them, and when agent X is interrogated, query the DB with the current context embedding. Use tools like LangChain’s `RetrievalQA` or similar to automate splitting, embedding, and searching [46†L349-L358].

Hierarchical Reflection: Periodically, run a “reflection” pass using the LLM to summarize or re-organize memory. For example, at end of day or after key events, prompt each agent to write a summary: “Based on your recent memories, summarize what you learned and how you feel.” Store this summary back in the agent’s memory store (possibly at a higher level). Over time you get layers: raw dialogue » intermediate summaries » trait updates. Such hierarchical summarization compresses long interactions into compact

insights. (This echoes the “Reflection” component in generative agents [65†L29-L37] and concepts in MemGPT [70†L138-L147] .)

Context Compression & Token Budgeting: To keep each agent’s prompt under ~1.5k tokens, use a mix of RAG and summarization. Only include the tail of the actual conversation (say last 5–10 messages) plus top-N retrieved memory snippets [46†L339-L348] . Truncate or compress older conversation. For example, before each query, generate a short summary of earlier dialogue turns (or use a pretrained summarizer) and include that instead of raw logs. Alternatively, maintain a rolling summary memory that gets updated when the buffer grows too large [46†L339-L348] . Use the strategy recommended by Agenta: when approaching context limit, evict least important chunks via summary, and rely on retrieval for detailed facts [46†L339-L348] [46†L431-L440] .

JSON State Syncing (Function Calling): After each LLM chat or agent thought, we need to update the character’s structured JSON state (e.g. `{anxiety:80, trust:35, hasSecret:true}`). The safest way is to *have the LLM output JSON directly*, using a strict schema or function call interface. For instance, define a JSON schema (or OpenAI function) listing exactly the keys (anxiety, suspicion, etc.). Then instruct the model: “Return your answer in JSON format with these fields.” Using a structured-output approach ensures format consistency [51†L39-L48] [51†L81-L90] . For example, you might call the LLM with:

```
System: Update {name}'s state from the recent dialog. Output a JSON object like
{"anxiety":INT, "trust":INT, "secretRevealed":BOOL, ...}. If nothing changes,
leave values unchanged.
User: [recent dialog content...]
Assistant: {...}
```

Validate the JSON (e.g. with a JSON schema or Pydantic) before applying. This avoids having to parse free-text. Newer models (GPT-4o, Claude 3.5, etc.) generally respect strict JSON prompts well [51†L39-L48] . If a value seems nonsensical, you could double-check with a short secondary prompt or clamp it. The key is to rely on the model’s structured output mode (e.g. OpenAI’s function calling) rather than having it freely narrate the state.

3. Narrative Director & Mystery Mechanics

Director Agent: Build an asynchronous “Game Master” process that *monitors the global state* and triggers events to steer the narrative. Architect it as another agent or service running off the main loop. It periodically polls character states (e.g. aggregated trust/conflict levels) and the player’s progress. When it detects stagnation (e.g. no clue found in many ticks) or rising tension (stress metrics reaching thresholds), it “injects” events: power outages, leaked evidence, surprise witnesses, etc. These events can be scripted or LLM-generated. For example, feed the Director context (“Scene: Detective in interrogation room with these NPCs; tension is high; no progress”) and prompt it to create an event (“Generate a dramatic interlude or hint”). The Director’s triggers can be rule-based (if `time_since_clue > X` or `sum(anxiety) > Y`) and its content generative. This loosely follows the idea of a “narrative manager” from interactive storytelling systems [48†L133-L142] [48†L200-L209] .

Deception and Secrets: Model lies by giving each character a *base honesty or deception bias* and *hidden knowledge*. Keep a flag like `knowsSecret` and `willLie` in the JSON state. When questioning, include instructions that the LLM should respond truthfully or deceptively based on those flags. For instance, in the prompt: "Character has been lying. Respond with a convincing but false answer without revealing your discomfort." Use the `anxiety` or `trust` variables: if `anxiety` or `stress` is high, instruct the model to possibly crack or slip a secret. You might do it in two steps: first ask "What is {X} asked?" and then "Given that {character} trusts the player only by TRUST%, how honestly should they answer? Output a final statement." Embody contradictions by having the Director or a logic check detect mismatches. If the model says "I never met X" but memory says they did, your system should catch it: either adjust the trust (uncovered lies) or force consistency (e.g. "Actually you did meet"). One approach is to occasionally run an LLM "consistency check" between the log and JSON: e.g. call a model with "Based on {character}'s known history and the statements they made, update their truthfulness rating."

Prisoner's Dilemma Dynamics: To simulate cooperation/betrayal among NPCs (especially if interrogated separately), maintain pairwise "trust" scores. After each conversation involving pairs (even indirectly), adjust these. For example, if the player pressures one NPC about the other, the first may "betray" the second by default (lowering mutual trust) or choose loyalty if their AI prompt suggests it. You can treat it like a PD payoff matrix: if both have incentive to deny knowledge of each other's secrets, they "cooperate" (both lie), but if one thinks betraying benefits them, they "defect." Internally, use the JSON state: when one NPC discloses evidence to the player about another, set a flag that the betrayer's `guilt` changed, and decrease mutual trust. Future dialogues should account for that changed state. In code, you might do a simple check: after Player interrogates A about B, have A's next turn consider "should I mention B's secret?". If we model a utility or threshold on loyalty, update JSON and tag that event. This essentially embeds a one-shot PD with evolving payoffs based on player actions.

4. API Routing, Cost Optimization & Multilingual Models

Tiered Model Pipeline: Use cheaper models for background tasks and higher-end for player-facing tasks. For instance, use a small/fast model (e.g. a Claude 3.5 Mini or Mistral-7B) to parse dialogue and update JSON state, or to vectorize text for memory storage. Use medium models (GPT-3.5 Turbo, or Qwen 2.5-13B) for routine NPC-to-NPC chats or planner steps. Reserve top-tier models (GPT-4o, Claude 3.5 Sonnet) for critical narrative generation (suspenseful dialogue, player interrogation responses, creative hints). This **fallback chaining** strategy is echoed in rate-limit designs: if GPT-4 is near quota, shrink prompts or degrade to GPT-3.5, etc. [61†L252-L262]. On OpenRouter, you can configure such a router: e.g. route short JSON updates to a cheap 7B model, and long-form dialogue to Claude Sonnet or GPT-4o. This saves costs and balances latency.

Language Benchmarks: For Persian (Farsi) and English, pick models known to handle these languages well. OpenAI's GPT-4o and Claude 3.5 Sonnet have strong multilingual support. Preliminary tests (user reports and benchmarks) suggest GPT-4/4o can handle Farsi nuance and idioms better than GPT-3.5. Qwen 2.5 72B (Alibaba) is also reported to perform competitively across languages, including Persian. You may want to test a few: for example, prompt "{text}" in Persian to each candidate model and check fluency. In general, route Persian dialogues to the model that gave best results in testing. (For example, use GPT-4o for high-fidelity Persian emotional dialogue, and GPT-3.5 for fallbacks.) There are no official OpenRouter Farsi benchmarks, but in our experience top-of-stack models (Anthropic Claude, GPT-4o) yield the most coherent, culturally aware output.

Rate Limiting & Async Queueing: When many agents run concurrently, implement a **central rate-governor** that tracks API usage. For example, use an asyncio Semaphore or token bucket to limit concurrent calls 【59†L120-L129】. Assign priorities: during critical player interaction, allow those LLM calls to preempt background tasks. One pattern is “**traffic-light throttling**”: monitor remaining quota and if low, automatically throttle or park low-priority agent calls 【59†L120-L129】. Another is a shared ledger: agents use a common token pool so idle capacity can be reallocated 【61†L189-L198】. In code, you might use a task queue (Celery, RQ, or custom asyncio) where each task claims “credit” from a shared pool before calling the API. On a 429 response, implement **exponential backoff with priority windows**: critical queries retry sooner, background queries wait longer 【62†L417-L425】. Finally, consider batching background updates in off-peak (e.g. during player thinking time) and preferring synchronous calls only when player input arrives.

5. Implementation Blueprint & Stack

Data-Flow Pipeline: A recommended sequence is:

1. **Player Input** arrives (text question or action).
2. Route to the relevant **Agent State Manager**: call the NPC’s LLM with current prompt (context + memories) to generate an utterance or answer. Use function calling to update that NPC’s JSON state from the dialogue.
3. **Memory Update**: take the player query + NPC response, summarize if needed, then embed and insert into memory DB. Also possibly trigger a “reflection” worker: e.g. on certain ticks, run an LLM to compress or augment memory.
4. **Director Evaluation**: in parallel or after the move, run the Director agent: check global state metrics (e.g. how many clues found, average stress) and decide if an event should fire this tick. If so, insert that event into the world state (for example, broadcast an alert to affected NPCs, update environment variables).
5. **Next Dialogue Generation**: based on updated state, resume background ticks. Agents autonomously chat or move (call LLMs for each NPC’s next action) until the player next intervenes.

Tech Stack: A Python-centric stack is natural. Use **FastAPI** or Flask for the backend API. For data stores: a relational DB (e.g. PostgreSQL) plus `pgvector` extension can hold tabular NPC attributes and an embedding index. Alternatively, a dedicated vector DB like Qdrant or Pinecone stores memories. For messaging/pubsub, use **Redis** or **RabbitMQ** to broadcast events (e.g. player actions or Director events) to agent workers. For LLM calls, use an async HTTP library (e.g. `httpx`) with tasks or Celery workers to manage concurrency.

Frontend/UI: You can build the interrogations in a webapp (Next.js/React) or integrate into a game engine. For a narrative game feel, **Godot** or **Ren’Py** could serve as a UI, with an HTTP or Python binding to call the backend API. For example, Godot scripts can POST the player’s question to FastAPI and display the NPC’s JSON/utterance response. Alternatively, a Next.js chat UI can suffice for a prototype.

Example Code Snippets:

1. Agent State Manager & JSON Updater (Python):

```
# Example: Update agent state from a dialogue turn via OpenAI function call
import openai
```

```

def update_agent_state(agent_id, recent_dialog, current_state):
    schema = {
        "type": "object",
        "properties": {
            "anxiety": {"type": "integer"},
            "trust": {"type": "integer"},
            "secretRevealed": {"type": "boolean"}
        },
        "required": ["anxiety", "trust", "secretRevealed"]
    }
    prompt = (
        f"Given the conversation {recent_dialog} and current state {current_state}, "
        f"update the JSON state of the agent. Respond with JSON only."
    )
    response = openai.ChatCompletion.create(
        model="gpt-4o",
        messages=[{"role": "user", "content": prompt}],
        temperature=0
    )
    new_state = response['choices'][0]['message']['content']
    # Parse and validate JSON (omitted here for brevity)
    return json.loads(new_state)

```

This example calls the LLM with a structured prompt and parses its JSON output as the new state.

1. Memory Retrieval & Reflection Worker:

```

from pgvector.psycopg import connect

def retrieve_relevant_memories(agent_id, query_text, top_k=5):
    conn = connect("dbname=game user=player")
    cur = conn.cursor()
    embedding = get_text_embedding(query_text) # your embedding function
    cur.execute("""
        SELECT memory_text FROM agent_memories
        ORDER BY embedding <=> %s LIMIT %s;
    """, (embedding, top_k))
    docs = [row[0] for row in cur.fetchall()]
    return docs

def reflect_on_memories(agent_id):
    # Periodically run: summarize older memories into a reflection note
    conn = connect("dbname=game user=player")
    cur = conn.cursor()
    cur.execute("SELECT memory_text FROM agent_memories WHERE agent_id=%s",
        (agent_id,))

```

```

all_memories = " ".join(row[0] for row in cur.fetchall())
prompt = f"Summarize important lessons from these memories: {all_memories}"
summary = openai.ChatCompletion.create(model="gpt-4o",
messages=[{"role": "user", "content": prompt}])
# Save summary back to a special memory table
cur.execute("INSERT INTO agent_memories (agent_id, memory_text) VALUES (%s,
%s)",
            (agent_id, summary['choices'][0]['message']['content']))
conn.commit()

```

This worker fetches top memories via vector search and also periodically compresses the agent's history into a summary stored in the DB.

Production-Grade Considerations: Ensure each part is asynchronous and fault-tolerant. Use connection pooling for DB, retry logic for API calls, and wrap LLM calls with schema validation and fallback. Keep character JSON and memories durable so the simulation can resume after a crash. Use a scheduler (like Celery Beat or APScheduler) to trigger Director checks and reflection tasks at intervals.

By combining these modular approaches – a flexible open-source agent loop (e.g. FastAPI tick server), RAG-backed memory, structured state updates, and a supervisor “Director” agent – you can create a rich, emergent detective-story simulation. Focus on defining clear JSON schemas and using LLMs where creativity is needed, while handling rote management (databases, event logic, rate-limiting) in code. This hybrid architecture ensures **believable NPC psychology**, dynamic narrative pacing, and scalable performance, in both English and Persian.

Sources: Established agent frameworks and memory systems [65†L29-L37] [67†L279-L287] [70†L52-L60] [70†L123-L130] , retrieval/context management best practices [46†L340-L349] [43†L119-L127] , and multi-agent rate-limit patterns [59†L120-L129] [62†L417-L425] .