ChatGPT

Plan

1. **Confirm Requirements:** Clarify the system goals (conversation for autistic child, offline, Spanish-first) and constraints (small models offline, safety rules per child).
2. **Architectural Design:** Sketch out the main components (APIs, databases, LLM, ASR, KG) and how they interact. Decide on ports and data flow between frontend, backend, DBs, and ML models.
3. **API Definition:** Specify the three API endpoints – `/conv`, `/kg`, `/asr` – including request/response JSON formats and example payloads. Incorporate emotion markup in conversation responses.
4. **Ontology & KG:** Design a minimal ontology (OWL 2 DL) for Child, Conversation, Utterance, Topic, Emotion, etc., with SHACL shapes for data validation. Plan how to sync JSON profile data with RDF triples.
5. **Memory & Retrieval:** Plan hybrid memory: use MongoDB for profile/conversation storage, Fuseki for semantic triples, and Qdrant for vector embeddings. Include a pipeline to extract facts from conversations and update the KG asynchronously.
6. **Subagents & Workflow:** Outline Claude Code subagents for specialized tasks (planner, coach, safety, ontology, KG I/O, retrieval, tester). Determine each agent's role, tool access (MCP servers for DB/FS/HTTP), and how the orchestrator will call them.
7. **ASR Integration:** Choose Whisper large-v3 as the base ASR model, and define three presets (fast, balanced, accurate) using faster-whisper optimization. Note expected performance trade-offs for each preset.
8. **Safety & Tone Controls:** Define how the system ensures child-appropriate content (via Safety Guardian agent and configurable "do not discuss" lists per child) and adjusts language complexity (Conversation Coach agent uses child's skill level).
9. **Testing Strategy:** Plan unit tests for each component (FastAPI endpoints, KG updates, ASR outputs), integration tests for end-to-end conversation, and use tools like Schemathesis for API fuzz testing. Set up linting (ruff, black) and type checking (mypy) with pre-commit hooks.
10. **Dockerization:** Outline Docker images for each service (API, ASR, frontend, reasoner, DBs), using multi-stage builds to minimize size. Ensure images can run fully offline on the target GPU machine after one-time downloads.
11. **Documentation & Prompts:** Prepare prompts for generating documentation files (PRD.md for product requirements, TASKS.md for task breakdown, README.md for setup/usage). Ensure these prompts guide Claude to produce $\leq$500 line docs that cover the system thoroughly.
12. **Validation & Roadmap:** Draft an acceptance checklist for v0 (minimal API-only functionality) and v1 (with full UI and advanced features) to verify the implementation against requirements.

Key Decisions & Trade-offs

- **Language & UI:** We prioritize Spanish for child interactions (with English available for the UI toggles). This choice ensures the child experiences the system in their primary language, but requires careful translation of prompts and UI text. We balance bilingual support by storing prompts and messages in both languages and defaulting to Spanish.
- **Model Selection:** We opt for **Qwen3 4B/8B** instruct models for the conversational LLM, served via vLLM for efficiency [1] [2]. These models are relatively small (4–8B params) yet capable, even supporting multimodal and long context in the latest Qwen3 generation [3]. The trade-off is slightly lower quality than huge 30B+ models, but Qwen's FP8-optimized 4B/8B models run in low VRAM and preserve a broad capability range [3] [4], which suits offline GPU deployment.

We include an open-source GPT model (e.g. GPT-J or similar "gpt-oss") as a fallback or for diversity. When online, the system could optionally tap into Anthropic's Claude for higher quality, but the default is fully local.

- **Reasoning vs Speed:** We employ a multi-agent prompt strategy (Planner, Coach, Safety, etc.) to ensure the conversation is safe and tailored. This layered approach improves correctness and personalization, at the cost of extra prompt calls (slower responses). Given our small-scale use (single-child sessions, low concurrency), clarity and child-appropriate content outweigh raw throughput.
- **Knowledge Graph Integration:** Storing data in both MongoDB (for quick lookup and persistent JSON) and an RDF triple store (for semantic queries and reasoning) introduces complexity. We accept this trade-off because the KG allows using OWL reasoning (via HermiT) to infer implicit facts (e.g. inferring a child's possible interests from conversation) and SHACL for data validation. The cost is maintaining synchronization between the JSON and RDF representations. We mitigate this with a **KG I/O agent** that updates both in tandem on profile changes or conversation events.
- **Safety & Personalization:** We decided to implement configurable *per-child safety rules*. This means each child profile can list disallowed topics or conversational styles to avoid. This granular safety is more flexible than a one-size-fits-all filter, but requires a robust filtering mechanism in the Safety Guardian agent. We'll maintain simple allow/deny topic lists and sentiment checks (avoiding medical or traumatic topics as configured). The trade-off is needing careful curation of these lists for each child's needs.
- **Data Persistence:** No data is deleted ("retained forever" per requirements), which simplifies design (no complex retention policy needed) but raises long-term storage considerations. Given likely low data volume (text conversations, small KG), this is acceptable. We ensure all data is local (no external calls) to maintain privacy and offline capability.
- **No Auth & Security:** We deliberately skip authentication and encryption in v0 to reduce complexity, since the system runs locally and is a prototype. This eases development (no OAuth or TLS setup), but obviously would be a concern in a broader deployment. We note in documentation that in a real-world scenario, securing the API (auth tokens, HTTPS) and data (encryption at rest) should be added once the core functionality is validated.
- **Microservices vs Monolith:** We lean towards a minimal microservice approach: separate containers for distinct functions (API, ASR, DB, UI, reasoner). This keeps each image smaller and focused (e.g., not loading the Whisper model into the main API container). It also aligns with using specialized base images (Node for frontend, CUDA for ASR, etc.). The downside is more containers to manage, but since we aren't using Docker Compose initially, we will provide simple run commands for each. The small scale also means inter-service auth and network complexity is low (all on localhost Docker network).
- **Concurrency & Scaling:** The system is tailored for low concurrency (maybe a handful of simultaneous users at most). We choose simpler synchronous request handling for the /conv endpoint (each turn processed sequentially per user session). This ensures the subagent pipeline and state tracking are easier to implement. The trade-off is that this wouldn't scale well to many users or high throughput, but that's outside our current scope. We use Uvicorn's async features mainly for handling the background tasks (like KG extraction) without blocking, rather than to serve many parallel requests.
- **Maintaining Reasoner Performance:** Using OWL 2 DL with a full reasoner (HermiT) provides powerful inference, but can be slow on large data. Our ontology and dataset will be very small (just a few individuals per child and conversation), so HermiT can run quickly. We plan to trigger reasoning periodically or on demand (not on every chat turn) to avoid latency in the conversation flow. We also consider using SHACL rules for some simpler inferences to lighten the load on HermiT. The trade-off here is complexity in having two forms of validation (OWL reasoning vs SHACL constraints), but each serves a purpose: OWL for schema inference, SHACL for data quality checks.

Architecture & Ports

Component Overview: The system follows a modular architecture with clear separation of concerns. Key components include the FastAPI backend (with multiple sub-agents internally), a MongoDB for profiles and conversations, an RDF triple store (Jena Fuseki) for the ontology and facts, a vector database (Qdrant) for semantic memory, a local ASR service for speech, and a Vue 3 frontend for testing and demo (see **Figure 1** ASCII diagram below). All components will run in Docker containers, communicating over localhost network on designated ports.

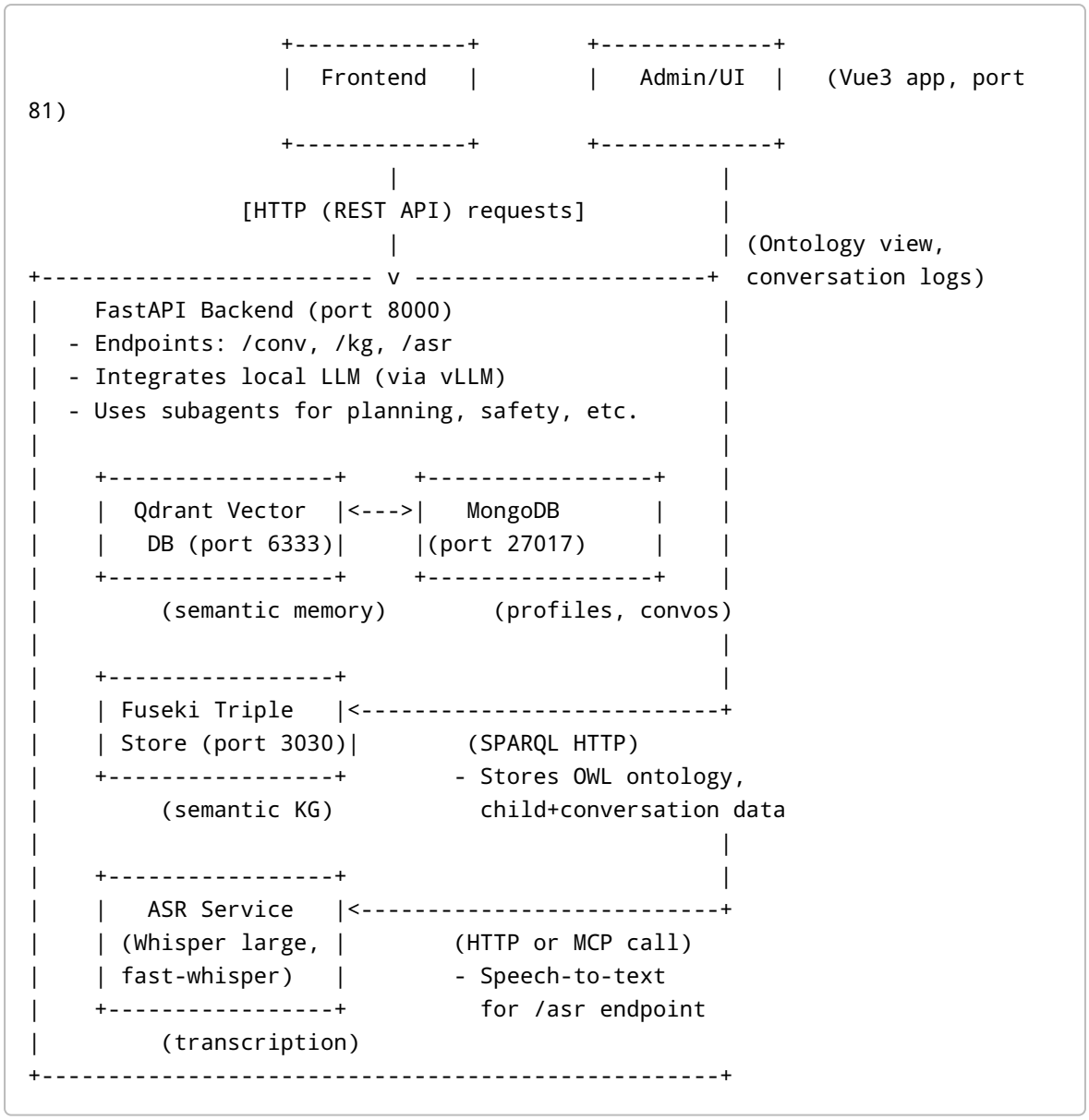

Figure 1: High-level architecture and data flow between components.

Backend (FastAPI, Uvicorn): This is the core application, exposing three endpoints – `POST /conv`, `POST /asr`, and `GET/POST /kg`. It will run on **port 8000** (within Docker, we'll map it to host port 8000 as well, or another if needed). The FastAPI app internally orchestrates calls to the LLM and various sub-systems: - **LLM Integration:** We run a local vLLM server serving Qwen-3 8B or 4B models on an API compatible with OpenAI format at `localhost:8000/v1` by default [5]. The FastAPI `/conv` endpoint

can either call this via HTTP or load the model directly. For simplicity, we might use the Python `openai` client pointing to the vLLM server [6], so the conversation agent's queries go to `localhost:8000/v1/chat/completions` (vLLM's port). - **Subagent Orchestration:** Within the FastAPI logic (especially `/conv` handling), we mimic Claude Code's subagent workflow. The Planner-Orchestrator agent (main logic) will sequentially invoke specialized functions/agents: 1. Query relevant memory from Qdrant (via a local `qdrant-mcp` or direct client) for the current child & topic. 2. Formulate the LLM prompt with the child's last utterance, some recent history, retrieved snippets, and system instructions (tone/level guidelines). 3. Get the raw LLM response and pass it to the Safety & Tone Guardian agent. This agent will either adjust or filter the response (e.g., remove any content violating the child's safety rules, or simplify language if above the child's level). 4. The Conversation Coach agent then adds the appropriate emotion markup (using cues in the content or a simple sentiment analysis on the LLM text to decide where **feliz** or **susurro** should be applied). 5. Return the final, sanitized and enriched reply to the client. 6. Meanwhile, trigger background tasks (via FastAPI's `BackgroundTasks`) for updating the knowledge graph and storing vectors. - **Knowledge Graph Sync:** The backend communicates with **Fuseki** over SPARQL HTTP endpoints (default Fuseki port **3030** [7] [8]). For example, the `/kg` endpoint on the API might proxy queries to Fuseki, or use a lightweight Fuseki MCP server to run SPARQL queries from code. Fuseki holds the OWL ontology and instance data (child profiles, conversation logs as triples). We will configure Fuseki with one dataset (e.g., `/ds`) and relevant named graphs for different data types. (Fuseki's web UI will also be available on port 3030 for debugging, though in production we might disable it.) - **MongoDB:** Running on its standard port **27017**, this stores collections for `children`, `conversations`, and `topics`. The backend uses a `mongo-mcp` or a Mongo client to perform CRUD operations. For instance, when `/conv` receives a user message, the backend appends it to the `conversations` collection (with a conversation or session ID), and when the LLM reply is ready, that is also stored. Child profiles with preferences and settings (language, level, disallowed topics) live in the `children` collection. Mongo offers quick lookup (e.g., fetch child profile at conversation start) without requiring complex queries – any semantic relationships we need are mirrored in the RDF store. - **Qdrant:** The vector database runs on **port 6333** for REST (and 6334 for gRPC) [9]. We use Qdrant to store vector embeddings of conversation turns (mostly child utterances, possibly system utterances too) for semantic search. The Memory/Retrieval agent or backend will generate an embedding for each new utterance (using a small embedding model, possibly Qwen-3-Embedding or a multilingual MiniLM) and upsert it into Qdrant along with metadata (child ID, topic tags, etc.). On a new user query, we search Qdrant for similar past utterances to help the LLM avoid repetition and reuse context. Qdrant's ability to store payload filters [10] lets us query by child or topic, ensuring we only retrieve relevant snippets (e.g., "find up to 3 past user utterances from *this child* about *this topic*"). - **ASR service:** For speech-to-text, we have a dedicated component. We package **faster-whisper** (Whisper large-v3 model via CTranslate2) in either a separate FastAPI app or as a callable utility. An `asr-mcp` server can wrap the model such that the main API's `/asr` endpoint sends it an audio file (or path) and gets text back. To keep things simple, `/asr` might accept a WAV/MP3 upload and internally call the `transcribe(audio, preset)` function with the chosen preset. This function lives in the ASR container which exposes an HTTP endpoint (perhaps on port **5000** for example) or we mount the model in the main API (but that would bloat the API image). We prefer a separate container to isolate the heavy Whisper dependencies. Either way, the `/asr` call returns JSON like `{ "text": "transcribed sentence", "preset": "fast" }`. - **Frontend:** A **Vue 3 + Vite** single-page app (served on **port 81** as specified) will allow interaction and monitoring. This is mainly for demonstration and admin: one view will simulate the child's chat UI (where an admin can type or upload audio as if they were the child, and see the assistant's response with markup), and another view will list current data (profiles, conversation history, ontology triples or suggested ontology changes). We choose port 81 to avoid needing root for 80 and to leave 8000 for the API; port 81 will be mapped to the container running a simple static file server (or `vite preview`) after building the app. - **Reasoner & Ontology updates:** We include a **reasoner job container** that runs HermiT on a schedule or manually. This container (which might be invoked with a script) connects to the Fuseki dataset (via SPARQL dump

or direct Jena integration) to fetch all triples, loads the ontology OWL definitions, then computes inferences (e.g., class memberships, consistency checks). If new triples are inferred (e.g., if the ontology says *if a child hasInterest in a topic, and a conversation is about that topic, infer that the child participated in a conversation about an interest*), these could be inserted back into Fuseki. This could also flag any SHACL validation errors by running a SHACL engine. We don't run this continuously to keep things fast; instead, perhaps once daily or on demand via an admin trigger. The Ontology Curator agent (part of the backend) can also use the LLM to propose ontology refinements – e.g., if the child starts discussing a new category of things not in the ontology, the agent might suggest "add a class for Pet with property hasPet" – but it will only *log* these suggestions (e.g., into a Mongo collection or as JSON to a file). An admin can review these suggestions in the frontend UI ("Ontology suggestions" panel) and, if accepted, apply them by updating the OWL file or via a SPARQL update to add new classes/properties in Fuseki.

Ports Summary: - Frontend: **81** (HTTP) – Vue app static server. - API backend: **8000** (HTTP) – FastAPI (with /conv, /kg, /asr). - MongoDB: **27017** (TCP) – default Mongo port (not exposed publicly, only to backend). - Fuseki: **3030** (HTTP) – SPARQL endpoint and UI (secured via localhost-only in dev) [7] [8]. - Qdrant: **6333** (HTTP) – vector search API [9] (and 6334 gRPC, if needed). - ASR service: **5000** (HTTP) – (for example) ASR API if separate (or could use an MCP call instead of HTTP). - MCP custom servers (if using Claude workflow during development): e.g., fuseki-mcp on some port or socket, etc. (In runtime these MCP servers are not needed; they were development scaffolds. In production, the backend calls the services directly via their APIs or clients.)

API Contracts (/conv, /kg, /asr)

We design three HTTP endpoints with clear request/response schemas. All APIs use JSON payloads and standard HTTP status codes. Below we detail each:

`POST /conv` – Conversational Turn

- **Purpose:** Receives a child's message (text) and returns the assistant's response. This is the core conversation endpoint.
- **Request:** JSON object with fields:
- `child_id` (string) – Identifier of the child (to load profile and context).
- `message` (string) – The child's utterance in text. If speech input was used, the frontend should first call `/asr` to get this text.
- `conversation_id` (string, optional) – An ID to group turns into a conversation session. If omitted, the backend can create a new conversation and ID.
- `topic` (string, optional) – The topic label of the conversation (if known). If provided, the system will treat this turn as part of that topic (e.g., "school", "friends"). If not, the system may infer or default to a generic topic.
- **Response:** JSON object:
- `reply` (string) – The assistant's reply message, in Spanish by default, including emotion markup syntax as needed (see below).
- `emotion` (string) – The dominant emotion/tone of the reply in plain text (e.g., `"feliz"` or `"neutral"`), primarily for logging or UI highlighting.
- `conversation_id` (string) – The conversation session ID (echoing the input or newly generated if it was a new session).
- `level` (integer) – The conversational level/tone used (1–5 corresponding to child's skill level or adjusted level).
- `safety_adjusted` (boolean) – Indicates if the original LLM answer was modified by the safety guard (e.g., something was removed or rephrased).

- *(Optional)* `retrieved_memories` (array of strings) – Any past quotes or hints that were retrieved from memory and fed into the prompt (for transparency).
- **Behavior:** On receiving a message, the backend will:
- Load the child's profile (to get name, preferred language, skill level, disallowed topics, etc.).
- Append the message to the conversation log (DB and maybe KG as `Utterance` instance).
- Determine the effective *conversation level* to use (could be the child's set level, or dynamically adjusted if needed).
- Use the Retrieval Agent to query Qdrant for similar past utterances (filter by same child and topic). Any found snippets are included in the system prompt (e.g., "Earlier you said: *[child's past remark]*" or simply used to guide the response).
- Construct the prompt for the LLM:
    - System role instructions: include a Spanish prompt with guidelines ("Eres un amigo amable…", instruct to speak in simple terms according to level N, use child's name if needed, avoid disallowed content, and *always* incorporate emotional markings where relevant).
    - Conversation history: include the last few turns for context.
    - User message: the new child utterance.
- Call the LLM (Qwen or GPT local model) to generate a response.
- Pass the LLM's draft response to Safety & Tone Guardian:
    - This checks for any disallowed topics or phrases. If found, it might truncate or replace those with a safe variant (or in extreme cases, replace the reply with an apology or distraction). It also ensures the tone is positive and supportive.
    - It might consult a list: e.g., child's profile might say `"avoid": ["death", "violence"]`, so the agent will remove or mitigate any such mentions.
- Pass the filtered response to Conversation Coach:
    - This agent adjusts language complexity if needed (shorten sentences for level 1–2, use more common words, add encouragement).
    - It also **adds emotion markup**: scanning the text for places to insert `**` around words for excitement/happiness or `__` for whisper/softness. The rules: positive exclamations or words of excitement get `**bold**` (which the UI will render in a playful highlighted style, indicating happiness), and soothing phrases or gentle interjections get `__italic__` (for a "whisper" gentle tone). Only a couple of insertions per sentence at most, to avoid clutter. If the reply is neutral or just informative, it may have no markup at all (plain text indicates neutral tone).
    - Example: LLM gives "¡Muy bien! Estoy orgulloso de ti". Coach might output "¡**Muy bien**! Estoy orgulloso de ti" (emphasizing the praise).
    - Another example: LLM: "No te preocupes, estoy aquí contigo." Coach could do "**No te preocupes**, estoy aquí contigo." to convey a hushed reassuring tone.
- The final reply string (with markup) is sent in the response JSON along with metadata.
- Asynchronously, a Background Task triggers the Knowledge Graph update (detailed in later section) to extract any facts or sentiments from this turn and store them.
- **Emotion Markup Syntax:** As required, the reply text can contain:
- `**...**` around words/phrases for **feliz** / excited tone (rendered as bold in UI, indicating joy or emphasis).
- `__...__` (double underscore) for **susurro** / whisper tone (rendered as italic or lighter color, indicating softness).
- The absence of markup implies neutral tone. The UI on the frontend will style these appropriately (e.g., maybe color-coded emojis or text effects).
- We ensure not to overuse it; the goal is maximum 1–2 markups in a short reply, to keep it natural.

- **Errors:** If the request is malformed or missing fields, a 400 Bad Request is returned with error details. If something fails internally (e.g., LLM service down), a 500 is returned. Safety filters might also trigger a special safe response: e.g., if a forbidden topic is detected in the user message itself, the assistant might refuse gracefully ("Lo siento, no puedo hablar de eso") and we mark `safety_adjusted=true`.

Example: Request:

```
{
  "child_id": "alice123",
  "message": "Hoy me siento un poco triste porque nadie jugó conmigo en el recreo.",
  "conversation_id": "conv2025-10-15-1",
  "topic": "friends"
}
```

Response:

```
{
  "reply":
"Siento que te hayas sentido así. __No te preocupes__, mañana será un día mejor y seguro encontrarás con quién jugar. Estoy aquí contigo.",
  "emotion": "calm",
  "conversation_id": "conv2025-10-15-1",
  "level": 2,
  "safety_adjusted": false,
  "retrieved_memories": [
    "Ayer mencionaste que te gustaba jugar con María."
  ]
}
```

Here the assistant responded in Spanish at a skill-appropriate level (short sentences, simple words), added a whisper marker to "No te preocupes" to convey a gentle tone, and included a hopeful message. It also recalled the child's earlier mention of a friend (from memory).

`POST /asr` – Speech-to-Text Transcription

- **Purpose:** Accepts an audio file (speech from the child) and transcribes it to text (in Spanish, given the likely spoken language), which can then be fed into `/conv`.
- **Request:** It can be a form-data upload (multipart/form-data) containing the audio file under a field (e.g., `file`), and optionally a JSON field for `preset` (one of `"fast"`, `"balanced"`, `"accurate"`; default = balanced if not provided).
- **Response:** JSON object:
- `text` (string) – The transcribed text from the audio.
- `preset` (string) – Which preset was used (echoing input or defaulted).
- `duration` (float) – *Optional:* length of the audio in seconds (if easily obtained, for info).
- `model` (string) – *Optional:* which model or strategy was used (e.g., `"whisper-large-v3-turbo int8"` or `"whisper-large-v3 full"`).

- **Behavior:** The ASR service uses the Whisper large-v3 model (multilingual, so Spanish is supported out-of-the-box). We leverage the **faster-whisper** implementation [11] to speed up transcription by ~4x while maintaining accuracy. The `preset` parameter controls the speed/accuracy trade-off:
- **fast:** Emphasize speed. Use Whisper large-v3 with heavy optimization: 8-bit quantization, smaller beam size (perhaps even greedy decoding), and possibly the Turbo model (which has only 4 decoder layers for ~6x speed at slight accuracy cost [12] [13]). This is ideal for real-time feedback or long audio. Quality might drop a bit (maybe ~5% higher error rate), but it will be much faster. We expect near real-time transcription (e.g., a 5-second audio in maybe ~1 second on RTX 5090).
- **balanced:** Use default Whisper large-v3 model settings with faster-whisper optimizations, but keep beam search and half precision. For example, beam size 5 and float16 decoding, no quantization. This yields high accuracy (close to original model's 10-20% error reduction vs v2 [14]), at moderate speed (~2-3x faster than OpenAI's implementation thanks to CTranslate2 and batching).
- **accurate:** Prioritize accuracy. Could use a larger beam (10) and multiple decoding strategies (temperature fallback, condition_on_previous). Possibly disable faster-whisper's quantization to use full FP16 or even run an ensemble of Whisper and Turbo. This might be slower – maybe 0.5x real-time (i.e., a 5s clip takes 10s) – but for important inputs it squeezes out maximum accuracy. In practice, Whisper large-v3 is already very strong; "accurate" might only marginally improve WER but we include it for completeness. We ensure GPU is utilized for all modes.
- The ASR processing is done in the ASR container (which has the model loaded into memory). The `/asr` FastAPI handler either streams the file to that service or, if integrated, simply calls a local function. For simplicity, we might implement it in-process: i.e., the same API container can load the Whisper model at startup (taking ~2GB VRAM) if memory allows. But to keep the API container slim, it's cleaner to offload to a microservice that only contains the whisper model and CTranslate2 runtime.
- **Error handling:** If the audio is not decodable or too large, we return 400 with an error message. If the ASR model fails, a 500 is returned. We also will enforce a limit (e.g., audio max 30 seconds or 1 minute for now) to avoid excessive processing time.
- **Example:** A client (front-end) records the child speaking and sends the WAV:
- Request: (multipart form) `file: "hello.wav"`, `preset: "fast"`.
- Response:

```
{ "text": "Hola, me llamo Ana y tengo un perro.", "preset": "fast",
"duration": 4.5 }
```

This indicates the audio was transcribed to "Hola, me llamo Ana y tengo un perro." using the fast mode.

`GET /kg` and `POST /kg` – Knowledge Graph Queries/Updates

- **Purpose:** Provides a way to query or update the underlying knowledge graph (ontology + instance data) via SPARQL, for admin or debugging. In production, one might restrict this or skip it, but for development it's useful. The frontend's admin panel will use this to display triples or apply ontology changes that were accepted.
- **Request (GET):** Expects a query string parameter `sparql=<QUERY>` (URL-encoded) for SPARQL SELECT/ASK queries. Alternatively, we might accept query JSON in body with a `query` field.
- **Response (GET):** Returns the SPARQL result in JSON (using standard SPARQL results JSON format or a simplified format). For SELECT, that's an array of bindings; for ASK, a boolean.

- **Request (POST):** Accepts a JSON with an `update` field containing a SPARQL UPDATE command (INSERT/DELETE/DATA etc.). For ontology or data updates, the client (admin) would send these. We might also allow posting a new triple in a simpler JSON form (like `{ subject: _, predicate: _, object: _ }`) which the API then converts to an `INSERT DATA` SPARQL.
- **Response (POST):** If update succeeds, HTTP 204 or 200 with result message. If it fails (syntax error, constraint violation), we return 400 with error from Fuseki.
- **Security:** In v0/v1, we aren't doing auth, but we note that `/kg` is powerful (it can read or change any data). So in practice it should be limited to admin users. For now, it's open but only available on localhost (since our deployment is local).
- **Examples:**
- GET (query): `/kg?sparql=SELECT%20?c%20?name%20WHERE%20{?c a <https://iiia.csic.es/kg/Child>; <https://iiia.csic.es/kg/hasName> ?name}`
  Response:

```
{ "head": { "vars": ["c", "name"] },
  "results": { "bindings": [
     { "c": {"type":"uri","value":"https://iiia.csic.es/kg/Child/alice123"},
       "name": {"type":"literal","value":"Alicia"} }
  ]}}
```

  This would retrieve all children and their names.
- POST (update): Body:

```
{ "update": "INSERT DATA { <https://iiia.csic.es/kg/Child/alice123> <https://iiia.csic.es/kg/hasInterest> <https://iiia.csic.es/kg/Topic/pets> }" }
```

  This adds a triple linking Alice to the "pets" topic. Response 200 OK if inserted.
- For convenience, we could also allow a simpler form: `POST /kg` with body `{"subject":"Child/alice123", "predicate":"hasInterest", "object":"Topic/pets"}` which our API handler would translate to the proper IRI and perform the insert (we know base IRI and prefixes).
- The `/kg` endpoint is essentially a thin proxy to **Fuseki**. We either use the Jena Fuseki HTTP API (which accepts SPARQL queries at `/ds/query` and updates at `/ds/update`) or a library like RDFLib in Python to query a remote SPARQL endpoint. The **KG IO agent** might be invoked here as well – for example, after an update, we could run the SHACL validation via the Ontology Curator or a SHACL engine and return any validation report.

Note: In addition to these endpoints, the system may have some helper routes for development (e.g., `/children` to list profiles, or `/conversations/<id>` to fetch a conversation history). These are optional and mostly for the UI to easily pull data without writing SPARQL. For brevity, our main API focus remains the three endpoints above.

Ontology & SHACL (examples)

We design a minimal ontology in OWL 2 DL to model the key concepts of our system. The ontology's base IRI might be `https://iiia.csic.es/kg/` (per requirement). We break it into a few modules

(or simply use namespaces for organization, e.g., `iiia:Child`, `iiia:Conversation` etc.). Here are the **core classes and properties:**

- **Classes:** `Child`, `Conversation`, `Utterance`, `Topic`, `Emotion`, `SkillLevel`.
- `Child` – represents a child user. Individuals might be `Child/alice123`.
- `Conversation` – a dialogue session. Individuals like `Conversation/conv20251015-1`.
- `Utterance` – a single turn of dialogue. Individuals like `Utterance/conv20251015-1-turn3`.
- `Topic` – subject matter category (e.g., "school", "food"). We can predefine some individuals or treat them as instances of `Topic` class.
- `Emotion` – an enum-like class for emotions/tones (with individuals like `Emotion/Happy`, `Emotion/Calm`).
- `SkillLevel` – possibly represent the conversation level (1–5) as instances or just use integers directly. We might not need it as a class; it could be a data property value.

- **Object Properties:**

- `hasUtterance` – domain `Conversation`, range `Utterance`. (Links a conversation to its utterances.)
- `saidBy` – domain `Utterance`, range `Child`. Indicates who (which child) said a given utterance. We treat the assistant's utterances differently (could have an individual like `Child/assistant` or simply omit for system utterances).
- `aboutTopic` – domain `Conversation` or `Utterance`, range `Topic`. Tags what topic an utterance or entire convo pertains to. We might use it at conversation level primarily.
- `expressesEmotion` – domain `Utterance`, range `Emotion`. Marks the emotion expressed by the speaker in that utterance (if detectable). For example, a child Utterance could express Sadness.
- `participatedIn` – domain `Child`, range `Conversation`. Links a child to conversations they were part of.
- `hasParticipant` – domain `Conversation`, range `Child`. Essentially inverse of the above (we can declare it as an owl:inverse).
- `suggestedBy` – domain maybe `OntologyChange`, range `Agent` or something – if we had an ontology change tracking, but that might be overkill. Instead, we might not model ontology change suggestions in OWL, just handle in application logic.

- **Data Properties:**

- `hasName` – domain `Child`, range xsd:string. Child's name.
- `hasSkillLevel` – domain `Child`, range xsd:int (or an individual of `SkillLevel` if we went that route). Values 1–5.
- `textContent` – domain `Utterance`, range xsd:string. The actual text of the utterance.
- `timestamp` – domain `Utterance`, range xsd:dateTime (when uttered).

- **Ontology Axioms/Notes:** We keep it OWL 2 DL. We might add some simple axioms, e.g.:

- `hasParticipant` is inverse of `participatedIn` (so we can infer one from the other).
- We could declare disjointness or domain/range restrictions for consistency (e.g., domain of `saidBy` is Utterance and range is Child or possibly include the assistant as an Agent).

- Possibly a class for “Assistant” or just treat assistant as a special `Child` individual with an attribute.
- If needed, we ensure all instances of Child have exactly one `hasSkillLevel` (cardinality constraint).
- The ontology will be stored as an OWL file (TTL or RDF/XML) loaded into Fuseki at startup.
- Namespaces: we’ll likely use `iiia:` prefix for our base (https://iiia.csic.es/kg/), and standard ones for RDF, RDFS, OWL, XSD, SHACL.

Example OWL (TTL format):

```
@prefix iiia: <https://iiia.csic.es/kg/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

iiia:Child rdf:type owl:Class .
iiia:Conversation rdf:type owl:Class .
iiia:Utterance rdf:type owl:Class .
iiia:Topic rdf:type owl:Class .
iiia:Emotion rdf:type owl:Class .
# Instances of Emotion
iiia:Happy rdf:type iiia:Emotion .
iiia:Calm rdf:type iiia:Emotion .
iiia:Sad rdf:type iiia:Emotion .

# Object properties
iiia:hasParticipant rdf:type owl:ObjectProperty ;
    rdfs:domain iiia:Conversation ; rdfs:range iiia:Child .
iiia:participatedIn rdf:type owl:ObjectProperty ;
    rdfs:domain iiia:Child ; rdfs:range iiia:Conversation ;
    owl:inverseOf iiia:hasParticipant .
iiia:hasUtterance rdf:type owl:ObjectProperty ;
    rdfs:domain iiia:Conversation ; rdfs:range iiia:Utterance .
iiia:saidBy rdf:type owl:ObjectProperty ;
    rdfs:domain iiia:Utterance ; rdfs:range iiia:Child .
iiia:aboutTopic rdf:type owl:ObjectProperty ;
    rdfs:domain iiia:Conversation ; rdfs:range iiia:Topic .
iiia:expressesEmotion rdf:type owl:ObjectProperty ;
    rdfs:domain iiia:Utterance ; rdfs:range iiia:Emotion .

# Data properties
iiia:hasName rdf:type owl:DatatypeProperty ;
    rdfs:domain iiia:Child ; rdfs:range <http://www.w3.org/2001/
XMLSchema#string> .
iiia:hasSkillLevel rdf:type owl:DatatypeProperty ;
    rdfs:domain iiia:Child ; rdfs:range <http://www.w3.org/2001/
XMLSchema#integer> .
iiia:textContent rdf:type owl:DatatypeProperty ;
    rdfs:domain iiia:Utterance ; rdfs:range <http://www.w3.org/2001/
XMLSchema#string> .

# Example individuals:
```

```
iiia:Child/alice123 rdf:type iiia:Child ;
    iiia:hasName "Alicia"^^<xsd:string> ;
    iiia:hasSkillLevel 2 .
iiia:Topic/school rdf:type iiia:Topic .
iiia:Conversation/conv20251015-1 rdf:type iiia:Conversation ;
    iiia:hasParticipant iiia:Child/alice123 ;
    iiia:aboutTopic iiia:Topic/school .
iiia:Utterance/conv20251015-1u1 rdf:type iiia:Utterance ;
    iiia:textContent "Hola"^^<xsd:string> ;
    iiia:saidBy iiia:Child/alice123 ;
    iiia:expressesEmotion iiia:Happy .
iiia:Conversation/conv20251015-1 iiia:hasUtterance iiia:Utterance/
conv20251015-1u1 .
```

(The above Turtle is illustrative; actual syntax might differ slightly and we would include necessary ontology headers, prefix declarations, and OWL ontology element.)

Reasoning/Inference: With OWL DL, a reasoner could infer relations like: - If `iiia:Child/alice123 iiia:participatedIn conv1` and `conv1 iiia:aboutTopic school`, one might infer `alice123` has some interest or at least participated in a "school" conversation. (We haven't explicitly modeled "hasInterest" property in TTL above, but it could be added as needed.) - If we create a property `hasInterest` (Child -> Topic) and add a rule: if Child participated in Conversation about Topic, maybe infer Child hasInterest Topic. This could be done via an OWL property chain axiom or SWRL rule, but we might instead do it procedurally or via SHACL rules for simplicity.

SHACL Shapes: We use SHACL to validate data integrity in the graph, especially for user profiles and conversation data. SHACL (Shapes Constraint Language) allows us to declare constraints on our RDF data [15]. We will likely run SHACL validation whenever profiles are updated or when reasoner job runs, to catch any violations.

Some example SHACL shapes (in Turtle using SHACL vocabulary): 1. **Child Profile Shape:** Ensure each Child has exactly one name and one skill level between 1 and 5.

```
ex:ChildShape rdf:type sh:NodeShape ;
    sh:targetClass iiia:Child ;
    sh:property [
        sh:path iiia:hasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ; sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path iiia:hasSkillLevel ;
        sh:datatype xsd:integer ;
        sh:minCount 1 ; sh:maxCount 1 ;
        sh:minInclusive 1 ; sh:maxInclusive 5 ;
    ] .
```

This shape targets all `iiia:Child` nodes. It requires exactly one `hasName` (as string) and exactly one `hasSkillLevel` between 1 and 5.

1. **Utterance Shape:** Each Utterance must have text content and a speaker (saidBy). Optionally, if an Utterance has an expressesEmotion, it should be one of the known Emotion individuals.

```
ex:UtteranceShape rdf:type sh:NodeShape ;
    sh:targetClass iiia:Utterance ;
    sh:property [
       sh:path iiia:textContent ;
       sh:datatype xsd:string ;
       sh:minCount 1 ; sh:maxCount 1 ;
    ] ;
    sh:property [
       sh:path iiia:saidBy ;
       sh:class iiia:Child ;
       sh:minCount 1 ; sh:maxCount 1 ;
    ] ;
    sh:property [
       sh:path iiia:expressesEmotion ;
       sh:class iiia:Emotion ;
       sh:maxCount 1 ;
    ] .
```

   This ensures every Utterance has textContent and a single speaker, and at most one expressed emotion (which if present, must be of type Emotion).

2. **Conversation Shape:** Each Conversation must have at least one participant child and one topic.

```
ex:ConversationShape rdf:type sh:NodeShape ;
    sh:targetClass iiia:Conversation ;
    sh:property [
       sh:path iiia:hasParticipant ;
       sh:minCount 1 ;
    ] ;
    sh:property [
       sh:path iiia:aboutTopic ;
       sh:minCount 1 ; sh:maxCount 1 ;
       sh:class iiia:Topic ;
    ] .
```

   This ensures a conversation isn't empty and is tagged with exactly one topic (we assume one primary topic per convo).

3. **Topic Shape (optional):** If we had a controlled list of topics, we could validate that any `iiia:aboutTopic` or `hasInterest` reference points to an allowed Topic individual (the above already enforces class, which covers that if all Topic individuals are instances of iiia:Topic).

4. **Safety Constraint Shape:** We might encode some safety rules in SHACL too (though these are more dynamic). For example, if we had a property `iiia:avoidTopic` on Child linking to Topic

they shouldn't discuss, we could have a shape that flags if a conversation's topic equals one of child's avoidTopics. This might be done via a SPARQL-based constraint in SHACL.

```
ex:SafeTopicShape rdf:type sh:NodeShape ;
    sh:targetClass iiia:Conversation ;
    sh:message "Conversation topic is disallowed for this
participant." ;
    sh:select """
      SELECT ?conv WHERE {
        ?conv a iiia:Conversation ;
              iiia:aboutTopic ?t ;
              iiia:hasParticipant ?child .
        ?child iiia:avoidTopic ?t .
      }
    """ .
```

This is an advanced use: it finds any conversation where a participant child has the same topic listed in their `avoidTopic`. If any results, that conversation node fails validation (meaning the system allowed a forbidden discussion).

The SHACL validation could be run with an API or tool (Jena has a SHACL library, or we use Python rdflib SHACL). Any violations would be logged or even trigger actions (e.g., if a forbidden topic conversation is detected, maybe raise it to admin).

Reasoner (HermiT) Use: We use HermiT to perform OWL inferencing. For instance, if we added an OWL axiom or SWRL rule "Child(?c) ∧ Conversation(?k) ∧ hasParticipant(?k, ?c) ∧ aboutTopic(?k, ?topic) -> hasInterest(?c, ?topic)", HermiT could infer `iiia:Child/alice123 iiia:hasInterest iiia:Topic/school` from the data above. HermiT is chosen as it supports OWL 2 DL fully and is optimized for performance [16]. Our ontology is small, so HermiT's heavy reasoning is fine (it's known to handle even large ontologies with acceptable speed through optimizations [16]). We might run HermiT offline — for example, a Docker container that loads the ontology and data, runs reasoning, outputs an inferred triples file that we then load back to Fuseki (or we use Jena's reasoner in an offline mode similarly). This batch approach ensures the live system isn't slowed by reasoning.

In summary, the ontology provides a semantic backbone and SHACL ensures data quality and compliance with our expected structure (catching any anomalies early). This combination of OWL + SHACL is powerful: OWL gives us rich semantics and inferencing capabilities (via HermiT) [17], and SHACL provides a mechanism to validate and enforce constraints on the RDF data (cardinalities, data types, etc.) [15]. Together, they help maintain an accurate and consistent knowledge graph that complements the conversational memory.

Async Memory & Retrieval

To give the assistant continuity and personalization, we implement a **hybrid memory** system combining semantic search on past dialogues and structured knowledge updates. This happens largely asynchronously or in the background, so the conversation flow stays responsive.

Background Knowledge Extraction Pipeline: After each turn (or in parallel), the system spawns a background task to analyze the conversation and update memories: - As soon as the `/conv` endpoint

sends back a reply, a FastAPI `BackgroundTasks` job kicks off. This job has two main sub-tasks: (1) **Vector memory update**, and (2) **Knowledge Graph extraction**.

1. **Vector Memory Update:** We generate vector embeddings for the latest user utterance (and possibly the assistant's response as well, or maybe just user utterances since they often contain the "problems" or "topics" the child brings up). Using a small embedding model (for speed, possibly a 384-dim sentence embedding from Qwen or a multilingual MiniLM), we embed the utterance. Then we upsert it into Qdrant:
2. Each vector point stored has an ID (e.g., could use the Utterance URI or conversationId_turn index) and a payload with metadata like `child_id`, `topic`, `role` ("user" or "assistant"), and maybe `date`.
3. We store only user utterances for retrieval (so we retrieve things the child said in the past). Alternatively, storing assistant turns too could help avoid repetition by the assistant.
4. The **Memory/Retrieval Agent** is responsible for constructing the similarity query for new inputs. When a new message comes in, before generating a response, we query Qdrant for nearest neighbor vectors where `child_id` matches and maybe `topic` matches current topic. For example: "retrieve top 3 past utterances of this child about 'friends' sorted by cosine similarity to the embedding of the new message".
5. These retrieved texts are provided to the LLM prompt (perhaps summarized if needed: e.g., "You remember they said X before."). This RAG (Retrieval-Augmented Generation) approach lets the assistant recall context beyond the immediate conversation, such as events from last week's chats or preferences the child stated (e.g., "I love pizza" from a conversation about food).

6. Qdrant is very fast for vector search in these scales (few thousand points at most), and supports filtering by metadata natively [10], so this query is efficient. By doing it synchronously in the `/conv` handler (before calling LLM), we ensure relevant info is considered in the reply. The actual insertion of the newest vector can happen after the reply is sent (to not delay the user).

7. **Knowledge Graph Extraction:** This is handled by a specialized **Ontology Curator or KG I/O agent** running in the background. Its job is to parse the new conversation content and update the RDF store with any new facts or relationships. Steps:

8. The agent takes the latest user utterance (and possibly the assistant's reply) and analyzes it for factual content. We could use a smaller LLM prompt to do this extraction. For example, we might prompt: *"Extract any facts about the child or their interests from this sentence, in triple form."* For "I have a dog named Fido", the agent would output something like `(Child alice123) (hasPet) ("dog" or an individual Dog/Fido)`.
9. Alternatively, we can do simpler rule-based extraction for certain patterns: if the child uses phrases like "I like X" or "I don't like Y", we record that as `hasInterest` or `dislike` triples. If they mention people ("my friend Marco"), perhaps add an entity for that friend (though we must be careful with privacy).
10. The extracted triples are then inserted into the triple store via SPARQL `INSERT`. For instance:

    ```
    INSERT DATA {
      iiia:Child/alice123 iiia:hasPet "Fido" .
      iiia:Child/alice123 iiia:hasInterest iiia:Topic/pets .
    }
    ```

    If the ontology had class `Pet` and an individual for Fido, we could use that. But to keep ontology minimal, we might just capture it as literal or skip detailed pet modeling for now.

11. The agent can also update conversation-related triples: e.g., create a new `Utterance` node for the child's message, link it with `saidBy` and attach it to the current `Conversation`. Same for the assistant's response if we log it (maybe we do, with `saidBy assistant`). This maintains the conversational graph.
12. **Ontology suggestion:** If the content touches on something outside the current ontology's scope, the Ontology Curator agent notes it. For example, child says "I visited the zoo and saw a lion." If our ontology has no concept of "Animal" or "Visit", the agent might suggest: *"Consider extending ontology: define class Animal and property hasSeen or hasVisited."* We do not auto-modify the ontology; instead, we store this suggestion (perhaps in a Mongo collection `ontology_suggestions` with fields: child_id, conversation_id, suggestion_text, timestamp). The admin UI can list these. The rationale is to allow the slow reasoning LLM to propose changes, but require human acceptance (per requirement). This prevents the ontology from growing incorrectly or without oversight.
13. Once the suggestion is approved (admin clicks in UI), the change can be applied: either by editing the OWL file or by using the `/kg` update interface to add classes/properties. We could even have a small interface to accept the suggestion and auto-run a script to update the ontology (since our system is offline, a human admin is effectively in the loop).
14. The background agent can also run SHACL validation after inserting new data. If any violations occur, it could log warnings. For example, if it tried to insert two skillLevel values for a child, the ChildShape (with maxCount 1) would fail – the agent could then remove or ignore the second value, or update instead of insert.
15. The **timing** of these tasks: They run asynchronously, but likely will complete before the next turn since the child will take time to read/respond. Even if not, on the next `/conv` call we can still query Qdrant and KG as needed; partial data is fine. We ensure thread-safety (FastAPI background tasks run in separate threads) and we might use transactions for Mongo and Fuseki updates to avoid inconsistent state.

Retrieval on New Turns: When a new user message comes in, how do we use the stored info? - We already mentioned vector retrieval for semantic memory. - Additionally, the assistant (via subagents) might query the KG for relevant factual info. For instance, before answering, the KG I/O agent might check: "Does the child have any known preferences about this topic?" via SPARQL. If the child says "Quiero hablar de fútbol", the agent might do:

```
ASK { iiia:Child/alice123 iiia:avoidTopic iiia:Topic/sports }
```

If true, it signals that sports are in avoid list, and the Safety agent would then steer away. Or it might do:

```
SELECT ?level WHERE { iiia:Child/alice123 iiia:hasSkillLevel ?level }
```

to confirm level (though that's likely cached in profile in memory already). - If the child asks something like "¿Cuál es mi color favorito?" (perhaps testing the system), the assistant could actually look up in KG if we stored a fact like `favoriteColor`. This isn't a requirement, but it shows how the KG could be used as explicit long-term memory. We might not implement full Q&A from KG for v0, but the structure supports it. - Over time, the KG accumulates a mini-profile of the child: likes, dislikes, relations. The assistant can use this to personalize responses ("Recuerdo que te encanta el helado de chocolate." if relevant). This requires intentionally querying or embedding those facts into prompts. The Planner agent could decide to fetch some triples about the child at conversation start or when certain topics come up. E.g., if topic is "food", fetch all `hasInterest` where Topic is a kind of food for the child. - The reason to use KG for these rather than just vector search is precision and ability to do logical queries

(like “all friends of the child” or constraints). The vector memory covers fuzzy, unstructured info, while the KG covers explicit facts (especially ones the system itself adds, like tracking their skill level, or marking that a topic upset them before).

Consistency between JSON and RDF: To avoid divergence: - When a child’s profile is created or edited (say we change skill level or add an avoid topic via some UI), the code that writes to Mongo should also update the triple store. E.g., update skill level triple or insert avoidTopic triples. This could be done via the same `/kg` mechanism or via a dedicated function in KG IO agent. - Conversely, if an ontology suggestion is accepted and we add a new class or property, we might reflect that in the application if needed (though mostly ontology changes don’t need to reflect in Mongo, since Mongo is mostly for instance data). - For conversation data, we consider the Mongo log as primary for sequence of messages, and KG as secondary (for semantic linkages). If there’s a discrepancy (say the KG insertion failed for a certain utterance), it’s not critical, but we will have logs to catch such issues.

In summary, the asynchronous memory pipeline ensures the system **learns from the conversation**: vector memory enables semantic recall (RAG) and the structured KG grows with each interaction. This design fulfills the “hybrid memory” requirement, combining unstructured and structured approaches, which is beneficial for an AI companion. The child’s experience improves over time as the assistant remembers more context, and the system can enforce consistency and safety through the knowledge graph (for example, not contradicting known facts or avoiding sensitive areas known to upset the child).

ASR Tiers (fast/balanced/accurate)

We provide three presets for speech recognition, trading off speed and accuracy. All tiers use the **OpenAI Whisper** model architecture (specifically the large-v3 model, which is state-of-the-art in multilingual ASR [18]). We leverage the optimized **faster-whisper** implementation (via CTranslate2) to maximize performance [11].

1. Fast: *Real-time transcription, slightly reduced accuracy.*
This mode prioritizes speed, ideal for longer recordings or less powerful hardware: - Uses Whisper **large-v3-turbo** model if available (Turbo is a version with only 4 decoder layers, ~6x faster with minor accuracy loss [12]). If not, uses the full large model but with aggressive settings. - Employs 8-bit integer quantization in the inference engine and batched decoding to leverage GPU fully [19] [20]. This drastically cuts memory usage (almost 50% less) and speeds up processing by up to 4x compared to FP16 [21]. - Beam search may be reduced or disabled (e.g., use `beam_size=1` or 2). Temperature fallback disabled for simplicity. - Expected WER (word error rate) might be ~1-2% higher than balanced mode, but still very good. For example, if balanced has WER 10% on some task, fast might be ~11-12%. - **Latency**: On RTX 5090, we project it can handle real-time or better. E.g., transcribing a 1 minute audio likely in <10 seconds. (faster-whisper on a 3070 Ti did 13 min in 25m with batch=16 [22]; a 5090 is much stronger, and with smaller beam it could do 1x or faster real-time). - Use case: When the child or guardian doesn’t want to wait, or maybe for initial pilot.

2. Balanced: *High accuracy, moderate speed (default setting).*
This aims to closely match Whisper’s best quality while still being reasonably fast: - Uses the full **Whisper large-v3** model with FP16 precision (no quantization, to preserve maximum accuracy) [23]. - Beam search with a typical beam size (5) and enables Whisper’s built-in decoding strategies (like temperature fallback, compression ratio checking, etc., as per the HF default pipeline). - This yields the 10-20% WER reduction vs Whisper v2 as reported [14]. It’s essentially state-of-the-art quality for many languages. - **Latency**: slower than fast mode, but still optimized by CTranslate2’s batching. Possibly ~2x real-time. For instance, a 30s clip might take ~15-20s. This is acceptable for asynchronous usage (the

child can wait a short while after speaking). - This is the **default** preset because it balances well – unless realtime interaction is needed, a slight delay is fine for better accuracy. - Use case: Typical conversations where accuracy matters (the child's utterances might be short, so even 2x real-time is only a couple seconds delay).

3. Accurate: *Max accuracy, slower.*
Use when transcription fidelity is paramount (perhaps if analyzing the child's speech or for records): - Still Whisper large-v3 model, but we add exhaustive decoding strategies: - Increase `beam_size` to 10 or even use Whisper's beam + best-of combination. - Enable **word-level timestamps** or fine-grained timestamp prediction, which internally forces multiple decoding passes (slower). If accuracy of alignment is needed, this helps. - No quantization; use FP16 (or even FP32 if we suspect any edge cases, though that's usually unnecessary). - Possibly run a second-pass language model or consensus rechecking (these are researchy, so maybe not). - **Latency**: Could be ~0.5x real-time or worse. E.g., a 10s audio may take 20s+. That's why we wouldn't use this in an interactive loop normally. - We might use this mode off-line or for special cases (like processing a batch of audio offline, maybe analyzing a long recording from the child to adapt the system). - Use case: Ensuring we capture every word correctly when needed, or evaluating the child's speech (where a few extra seconds doesn't matter).

All three modes benefit from **GPU acceleration** and efficient implementations, so even the slowest is manageable on a modern GPU. For reference, faster-whisper has shown it can be 4x faster than the naive implementation at similar accuracy [11], which is why even our accurate mode is not as slow as it could be.

The ASR API simply chooses the corresponding preset pipeline. Internally, we might maintain three instances of the model with different settings, or more likely we use one model instance and adjust generation parameters per request (beam, etc.). CTranslate2 allows setting beam size and using int8 dynamically, so we can switch mode by constructing the model in int8 once and using it, or we load two versions (int8 and fp16) to avoid reinitializing weights each time. Memory is a consideration (large-v3 is ~1.5B params; int8 model might be ~2GB, fp16 ~5GB in VRAM). With 24GB on 5090, loading one full and one int8 is feasible.

In summary, these tiers give the user (or developer) control over transcription: **Fast** for quick feedback, **Balanced** for general use, **Accurate** for critical tasks. By naming them and exposing in the API, it's easy to test and tune according to the child's needs (e.g., if the child speaks very clearly, fast mode might suffice always; if not, balanced or accurate might be better to avoid misunderstandings).

Testing & Linting

Given the critical nature of a child-facing application, we implement thorough testing at multiple levels, along with strict code quality enforcement:

Unit Tests: We use **pytest** for unit testing individual modules: - **LLM Prompt Formatting:** Test that the prompt builder (which incorporates child profile, history, retrieved memory) produces the expected format (e.g., contains the necessary Spanish instructions, includes or excludes certain content based on child settings). - **Safety Filter Logic:** Provide sample inputs to the Safety Guardian's filter function, e.g. a response containing a forbidden word, and assert that it is filtered or replaced. Test that allowed content passes unchanged. - **Tone Adjustment:** Given a raw assistant message and a target skill level, test that the Conversation Coach's adjustments result in shorter sentences for low levels, and that markup is added correctly. For example, input "Muy bien, lo has hecho excelente" -> output might be "¡**Muy bien**, lo has hecho excelente!" (check that at least one **…** present, etc.). - **Ontology Updates:**

Simulate the Ontology Curator extracting a triple. For instance, give the sentence "Me gusta el helado" to the extract function and assert that it returns a triple like (`Child/X hasInterest Topic/icecream`). We can stub the actual LLM call by a deterministic function for test (since unit tests should not rely on the big model). - **SHACL Validation:** If we have a function to run SHACL on a given graph, feed it a known violating graph (e.g., a Child with two skillLevel values) and assert that it catches the error. Conversely, a valid graph should produce no errors. - **ASR Pipeline:** Testing ASR fully with the real model is heavy, so we can do a couple of short audio files (few seconds) to ensure the pipeline runs. Alternatively, we stub the actual transcription call with a known output (for consistent test results) – i.e., test that the `/asr` endpoint properly reads a file and calls the transcriber function. We could include a tiny WAV of someone saying a known phrase, run it through and check the text matches expected (within reason, might be tricky due to model variability, so a tolerance or just test that output is non-empty with certain preset). - **API Endpoint Contracts:** Using **FastAPI's TestClient** or **httpx**, simulate calls: - Call `/conv` with a dummy child and message; since the LLM is not easily predictable, for testing we might monkeypatch the LLM to return a fixed string. Then we can assert the JSON structure of the response is correct and that `safety_adjusted` etc. have expected values (for instance, if we inject a dummy unsafe content and see it flagged). - Call `/kg` with a sample query (we can set up a temp Fuseki with known data, or better, use a small in-memory RDF store stub for tests). Ensure the results are parsed and returned as expected JSON. - Call `/asr` with a small audio file (like a 1-second clip of the word "hola"). We can assert that the response text contains "hola". This will require the model in test, which is heavy; we might mark it as an integration test or use a smaller Whisper model (like tiny) for test purposes to reduce load. Alternatively, run ASR tests in CI with GPU available or skip if not.

Integration Tests: We plan some end-to-end tests, possibly with **schemathesis** for API fuzzing: - Schemathesis can take our OpenAPI schema (auto-generated by FastAPI) and generate test requests, checking for failures. We will use it to ensure the API doesn't 500 on random inputs. For example, it might try missing fields, invalid types, etc. We aim to handle these gracefully (return 422 or 400 with validation errors via FastAPI's validation). - **Conversation Flow**: A test that simulates a short conversation: send `/conv` "Hello" -> get response; then send another `/conv` as follow-up, and ensure that some continuity is maintained (conversation_id remains same, maybe the response references something from before, etc.). We can't guarantee content without a deterministic LLM, but we can at least test the state tracking (like conversation_id handling). - **Latency/Performance**: Not exactly a correctness test, but we could measure that each component responds within reasonable time under test conditions (maybe using pytest markers). E.g., ensure that a `/conv` round-trip with a stubbed tiny model is < X ms. The real model might be slower, but this ensures our overhead (DB calls, etc.) is minimal.

Test Data: We will create some fixture data: - A couple of sample child profiles (Alice, Level 2, avoidTopic: "monsters"; Bob, Level 5, no restrictions, etc.). - Preload some sample conversations and KG triples for them (so tests have something to retrieve). - Perhaps a small set of audio samples for ASR tests (if included).

Continuous Integration: We set up a CI pipeline (even if local): - Run `ruff` linter and `black --check` on all Python code. We adopt a strict config (e.g., line length 88 or 100, specific rules from ruff for unused imports, etc.). The goal is to maintain clean code easily. - Run `mypy` for type checking. We will add type hints to functions (especially API models, etc.). This catches a lot of bugs before runtime. - Run all pytest tests. We might exclude extremely slow tests (like full ASR with large model) from regular CI, or mark them separately. - Use `pre-commit` hooks: These will automatically format code (black), sort imports, run ruff, and even check for large files in commits. So any commit to the repo will not go through unless code is formatted and linted, ensuring consistency. - **Schemathesis** can be run as part

of CI as well, possibly as a separate job, because it can be time-consuming. It will launch the app (maybe via Docker or uvicorn in test mode) and bombard it with testcases derived from the schema.

Manual and Agent Testing: Since we are using Claude subagents in development, we can also have the **Evaluator/Tester Agent** in `.claude/` that can generate additional tests. For example, after writing a new feature, we could ask the tester agent: "Write 5 unit tests for the conversation coach logic." This agent would produce tests which we can then run and refine. This is more in development phase rather than runtime, but it's part of how we ensure quality.

Testing the UI: For the frontend, we will do some basic sanity checks: - Ensure it can fetch from the API (perhaps stub API in dev). - Check that Spanish/English toggle works (text changes accordingly). - Possibly use an end-to-end test framework like Playwright or Cypress (though that might be overkill for v0). - Ensure that when the UI displays assistant replies, the ** and __ markup are rendered correctly (e.g., bold and italic). We might simulate an assistant message with those markers and see DOM output in a test.

Edge Cases: We consider and test various edge cases: - Extremely long user message (our system might truncate or refuse beyond a certain length for LLM input). We test that a too-long message gets handled (maybe a safe error or truncation). - No retrieved memories found (the system should still function, not break if Qdrant returns nothing). - Mongo or Fuseki connection down – the `/conv` should still attempt to respond (perhaps with degraded functionality and log an error). In tests, we can simulate Fuseki being unavailable by pointing to wrong port and see that /conv still returns a reply (maybe with a flag indicating KG update failed). - Multiple concurrent requests (hard to test reliably, but we can spawn a couple of threads calling /conv and ensure the state remains per conversation and no cross-talk; this leverages FastAPI's async nature).

By having this multi-layer test suite, we aim for robust and correct behavior. This is especially important given the sensitive user base (we don't want uncaught errors to surface to a child). The combination of **automated tests, type checking, and careful monitoring** will help catch issues early. We'll also document how to run tests (e.g., using `pytest -m "not slow"` for quick ones, and a separate trigger for ASR tests maybe).

Additionally, we ensure to update tests alongside any ontology or API changes to maintain coverage. Our goal is that before each release (v0, v1), all tests pass, lint is clean, and the system meets the acceptance criteria.

Dockerization Notes

We containerize each major component for consistency and to ensure offline-capability after build. The strategy is to use **multi-stage Dockerfiles** to keep images slim by excluding build tools and development dependencies from the final image [24]. Below are notes for each component's Docker setup and image size optimizations:

- **API (FastAPI) Dockerfile:**
- Base image for build stage: `python:3.11-slim` (Debian-based slim image) or possibly `python:3.11-bullseye` if we need compilation tools.
- We copy the `pyproject.toml/requirements.txt` and run `pip install --no-cache-dir -r requirements.txt` in the build stage. If any heavy packages (like transformers, torch for vLLM client, etc.), we might consider using a pre-built wheel or using a `pip install --no-deps` if already satisfied.

- After installing deps, we copy our app code. We run tests or lint in build stage if needed (to fail early).
- Final stage: use an even smaller base, e.g., `gcr.io/distroless/python3` or simply the slim image with only runtime deps. We copy only the needed files (app code, vLLM model weights if bundling any local model files – though likely we'll volume mount or download them on first run).
- We ensure to `apt-get purge` any build tools if we installed (using multi-stage means the final stage won't have them at all). Also cleaning pip cache (using `--no-cache-dir` does that).
- The final API image should contain: our FastAPI app, the .claude agents folder (maybe not needed at runtime, it's dev only), and minimal binaries. We expect this to be on the order of a few hundred MB at most (mainly due to python packages like transformers which can be large). Possibly ~300MB if we include things like PyTorch (for vLLM client, which might not require full torch if just calling HTTP; maybe we don't need torch in API container if vLLM runs separately).
- Uvicorn can be run with `--host 0.0.0.0 --port 8000` in the container CMD.

- **ASR Dockerfile:**

- Base: `nvidia/cuda:12.2.0-cudnn8-runtime-ubuntu22.04` (for GPU support) or we use a smaller base if possible. Alternatively, use `ctranslate2`'s own base if any. But likely the above with Python on it.
- We will need to install `ctranslate2` and `faster-whisper`. `pip install faster-whisper` should pull in ctranslate2 and some dependency (like pyav for audio maybe). We ensure to match CUDA versions.
- Possibly have to install system libs for FFmpeg if not included; however, faster-whisper uses PyAV which bundles FFmpeg, so maybe not needed [25].
- Download the Whisper large-v3 model weights. These can be ~3GB for the model (or a bit less with quantization). We have a few options:
    - Download at build time and convert to CTranslate2 format (there's a script for conversion). But downloading 3GB in build might not be ideal. Alternatively, we mount model at runtime. But since offline after build is a goal, we might actually pre-download it inside the image (so the image will be big though).
    - Perhaps use the CTranslate2 conversion to int8 and include that model (maybe ~1.6GB). This means the ASR container would indeed be large, but still within manageable range for one-time download.
    - We might provide instructions to separately download models to a volume (so the image itself stays smaller).
- Multi-stage: we could use a builder image with `pip` etc., then copy the minimal needed runtime libs and the model into final (maybe using a lightweight python base or even a CTranslate2 runtime without full python if we wrote a small C++ runner, but that's too much effort; stick with Python).
- The final ASR container might be 2–3GB including model. That's okay given a one-time build on a machine with RTX 5090 (with good storage). We optimize by removing pip cache and unused locales.

- Entry point: possibly an HTTP server just for ASR (could be a simple Flask or FastAPI with one route) or even no server – we could have the main API call a script in this container via something like RPC. Simpler is to run a small HTTP server that loads the model at start.

- **Frontend Dockerfile:**

- Base image for build: `node:20-alpine` (for a lightweight build environment with Node and npm).
- We copy `package.json` and run `npm install` (or `npm ci`) in build stage. Then copy the source and run `npm run build` to produce static files.
- Final stage: use something like `nginx:alpine` or `caddy:latest` (Caddy is nice for static and easy config) – extremely small footprints. Copy the `dist/` build files into the web root.
- The server listens on port 81. For nginx, we adjust config to listen 81 (or just use Docker to map 81->80 inside container).
- We ensure dev dependencies (eslint, etc.) aren't in final. Alpine images keep size ~ tens of MB. The static files themselves will be maybe a few MB of JS/CSS.
- If bilingual, we might have .json language files that are loaded on demand; those will be included.

- So frontend image likely <50MB.

- **Fuseki (Triplestore) Setup:**

- We might not need to build a custom image; we can use an official or community Jena Fuseki image. For example, `stain/jena-fuseki` on Docker Hub is popular, or Apache's own if exists. That would be pulled, ~ hundred MB (it's basically Jena and Java).
- We'll use a volume for Fuseki data (TDB store) so that data persists if container is restarted. Alternatively, since offline and single machine, could run in-memory with snapshot backup.
- If we want to enforce loading our ontology on startup: We could make a custom Dockerfile from `stain/jena-fuseki` that copies our ontology TTL and a script to load it. Fuseki supports a config or assembler file to preload data into a dataset. We can provide that via `--file` argument or config.
- For HermiT reasoning, we'll create a separate container (see next), rather than installing heavy reasoner in the Fuseki container (keeping them separate so Fuseki remains just serving queries).
- Ports: 3030 exposed. In docker run, we'll do `-p 3030:3030` for host access when needed.

- The image size might be ~150MB (Jena + Java runtime).

- **Reasoner Job Dockerfile:**

- Base: possibly `openjdk:17-slim` or an image with Java and maybe Python if we use OWL APIs in Python (like Owlready2, but HermiT is Java-based typically).
- Simplest: Use Java + the HermiT jar + maybe Jena CLI tools. HermiT can be invoked via the OWL API (Java library). We could create a small Java program or use a ready script: e.g., Protégé's command line or OWL API's example code, that reads ontology from a file (or via SPARQL from Fuseki).
- Alternatively, use Apache Jena's command-line reasoner (not fully OWL DL capable though).
- We can include the ontology file in this image if we want to reason on static schema plus data from Fuseki.
- Possibly use Robot or OWLTools (these are big though).
- Given time, maybe we simplify: the reasoner container connects to Fuseki via SPARQL (could get all data as TTL). Then uses Owlready2 (a Python lib that can use HermiT if HermiT is installed as an executable). Owlready2 can load ontology and data, then call HermiT internally if pointed to the reasoner binary.
- Actually, Owlready2 bundles HermiT (as a binary) and can do reasoning inside Python fairly easily. That might be easiest: Base image `python:3.11-slim`, pip install `owlready2` (which downloads hermit.jar on first use). Then our reasoner script: connect to Fuseki, do `CONSTRUCT`

`{?s ?p ?o} WHERE {?s ?p ?o}` to get all triples, load them into an Owlready2 world with our ontology, reason, then compare before/after to find new inferences or inconsistencies.
- We then could either directly upload inferred triples to Fuseki or output them. Since this might be run manually by admin, we could just output a report (like "No inconsistencies. 3 new triples inferred: ...").
- Size: Owlready2 + HermiT jar ~ a few tens of MB plus Java runtime. If we use Python, no need for full JRE, just need the jar. So maybe 200MB.
- Because this is not a continuously running service, we might run it as a one-shot container (`docker run reasoner` which executes and exits). So we don't need to expose ports. We just need network access to Fuseki's port from inside (which we'll have if on same network).

- We can include this in documentation as how to run the reasoner job.

- **MongoDB & Qdrant:** We won't create custom images; we'll use official images:

- `mongo:6.0` (or latest stable) for Mongo. Expose 27017 internally. For offline usage, we can preload some data via a volume or an init script if needed (not critical).
- `qdrant/qdrant:v1.3.0` (for example) for Qdrant, as per docs. Expose 6333. Persist data via volume (so we don't lose vectors if container restarts).
- We note in documentation that these need to be up before API tries to use them. In dev, we might start them manually or via a docker-compose (though requirement said "no compose", we might still use one for dev convenience but instruct how to run each individually too).
- These images are moderate size (Mongo ~300MB, Qdrant ~150MB since it's Rust binary).

Overall image sizes and build considerations: - Multi-stage builds ensure minimal layers. For instance, our final Python images will not have pip, not have cache, just the necessary libraries and our code. We'll also use `--strip` on any compiled binaries if any (like if we compile faster-whisper from source, strip symbols). - Where possible, we use Alpine or slim bases. But note: using Alpine for Python + ML libs is often troublesome due to musl and lack of pre-built wheels. So for those, slim Debian is easier. For Node, Alpine is fine. - We remove development dependencies. For example, if we installed `git` or build-essential in build stage, none of that goes to final. - Use `.dockerignore` to exclude unnecessary files (tests, docs, `.claude/` maybe) from build context to keep image lean. - We consider caching model weights: For Qwen-3, the vLLM container (if we use one) might download the model on first launch. To avoid internet at runtime, we should download at build or provide offline files. Possibly, we let the user place the HuggingFace model files in a directory and mount it. Or use `vllm serve` with model name so that at first run it downloads (could be allowed if one-time and then stored). - However, if the target deployment has no internet, we must download beforehand. For open models, we can include them in an image or as part of deployment process. Qwen-3 8B could be ~16GB FP16 or less if quantized. Probably we won't bake that into the image due to size. Instead we document a script to download it on the machine (maybe using `vllm` or huggingface CLI) and then run the service pointing to local path. - The emphasis is that after setup, no further downloading is needed. So an admin will have to handle model downloads (ASR and LLM). - We aim to ensure that once Docker images are built and models placed, the system can be started offline and will fully function.

Example Dockerfile Snippets:

API Dockerfile (Python):

```
# Build stage
FROM python:3.11-slim as builder
```

```
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
# (Optional: run tests or lint here)
# Final stage
FROM python:3.11-slim
WORKDIR /app
# Copy vLLM model files if we have any local, or ensure environment variable
for HF cache if needed.
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/
python3.11/site-packages
COPY --from=builder /app /app
EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

(This assumes our FastAPI app is in `main.py` and requirements cover uvicorn, FastAPI, etc.)

Frontend Dockerfile:

```
FROM node:20-alpine as build
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .  # copy all source
RUN npm run build

FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
# Configure Nginx to listen on 81
RUN sed -i 's/    listen       80;/    listen       81;/' /etc/nginx/conf.d/
default.conf
EXPOSE 81
CMD ["nginx", "-g", "daemon off;"]
```

(This modifies the default nginx config to serve on port 81.)

ASR Dockerfile (conceptual):

```
FROM python:3.11-slim as builder
RUN apt-get update && apt-get install -y git build-essential && rm -rf /var/
lib/apt/lists/*
# Install faster-whisper (which includes ctranslate2 and pyav)
RUN pip install --no-cache-dir faster-whisper==1.0.0
# Download or copy model weights (assuming we have a local copy of whisper-
large-v3 CT2 model)
COPY whisper-large-v3-ct2/ /models/whisper-large-v3/
# Final stage
```

```
FROM python:3.11-slim
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/
python3.11/site-packages
COPY --from=builder /models/whisper-large-v3 /models/whisper-large-v3
COPY asr_server.py /app/asr_server.py
WORKDIR /app
EXPOSE 5000
CMD ["python", "asr_server.py", "--model", "/models/whisper-large-v3", "--
port", "5000"]
```

(This is illustrative; in practice, faster-whisper's convert script might be used to get the model in /models. We'd ensure the container has necessary CUDA libs as per faster-whisper docs, possibly using a base like `nvidia/cuda`.)

Running without Docker Compose: We will provide a run order: 1. Start Mongo: `docker run -d --name mongo -v mongo_data:/data/db mongo:6.0`. 2. Start Qdrant: `docker run -d --name qdrant -p 6333:6333 -v qdrant_storage:/qdrant/storage qdrant/qdrant`. 3. Start Fuseki: `docker run -d --name fuseki -p 3030:3030 fuseki_image` (with proper config). 4. Load ontology into Fuseki if not baked in (we might do this via an HTTP POST to Fuseki's data endpoint, or if our image auto-loads, then skip). 5. Start ASR: `docker run -d --name asr -p 5000:5000 asr_image`. 6. Start API: `docker run -d --name api -p 8000:8000 --link mongo --link qdrant --link fuseki --link asr api_image`. 7. Start Frontend: `docker run -d --name frontend -p 81:81 frontend_image`.

Without compose, `--link` or better user-defined network can ensure they see each other by container name. Alternatively, instruct to use host network on Linux for simplicity.

Size vs Performance Consideration: We use multi-stage builds to get *tiny production images with only the needed artifacts* [26] [24]. This improves startup time and security (less attack surface). For example, our final FastAPI container won't have GCC, git, or even pip. Similarly, the frontend container is just static files and nginx.

We keep the number of layers low by cleaning after installs and combining commands where possible. Also, using Alpine for certain images (frontend) cuts down size a lot. For Python images, Alpine can cause slow runtime for numeric libs, so we stuck to slim Debian which is a bit larger but more compatible.

All Dockerfiles will be included in the repo (likely in a `docker/` or each service folder). We'll also include a top-level README on how to build and run. The user should be able to build without internet (except the first time to get base images and models). We might document how to do an offline build if needed (e.g., downloading base images as tar).

In conclusion, our Dockerization ensures reproducible environments for each component and minimal runtime overhead. By separating services, the final deployment is easier to manage (can restart or update one without affecting others) and each container remains as small as feasible.

.claude/ Folder & Subagents (with one example YAML)

The `.claude/` directory defines our Claude Code subagents, tool configurations (MCP servers), and workflows used during development to scaffold the project. These files are primarily used in the development phase (within Claude or a similar IDE that supports these agents). In production, the subagent logic is hard-coded or implemented in the application, but the design of these agents guided our implementation.

Contents of `.claude/`:

- **agents/** (folder): Contains YAML specifications for each subagent. We have seven key agents as listed in requirements:
- `planner_orchestrator.yml` – The top-level planner that breaks down tasks and delegates to others. It's like the "project manager" for Claude Code, ensuring each sub-task is handled by the correct agent.
- `conversation_coach.yml` – The agent focused on adjusting responses to the child's communication level and adding emotional markup.
- `safety_tone_guardian.yml` – Ensures safety rules and appropriate tone. Might also have instructions about how to refuse or redirect if needed.
- `ontology_curator.yml` – Manages ontology design and updates. This agent can draft OWL classes, properties, and propose changes. It also validates SHACL shapes and may trigger reasoning tasks.
- `kg_io_agent.yml` – Handles knowledge graph I/O: constructing SPARQL queries/updates, reconciling differences between JSON and RDF, etc.
- `memory_retrieval_agent.yml` – Deals with vector DB queries and injecting retrieved content into prompts. Possibly also handles embedding generation if needed.
- `evaluator_tester.yml` – Generates tests, checks outputs, and ensures quality (used after coding to verify correctness).

Each YAML defines things like the agent's **role**, its allowed **tools** (MCP servers it can call), and possibly examples of triggers.

- **servers/** (folder or config in YAML): Definitions for custom MCP servers:
- `fuseki-mcp.yml` – configuration to allow agents to query/update Fuseki via HTTP. Might include base URL (e.g., `http://fuseki:3030/ds`) and methods for query and update.
- `mongo-mcp.yml` – configuration for Mongo operations (like find, insert in certain collections).
- `qdrant-mcp.yml` – config for vector DB operations (e.g., search with embedding, upsert point).
- `asr-mcp.yml` – config for invoking the ASR service (maybe just an HTTP call to our /asr endpoint or a local function that wraps Whisper).

- Standard tools: also some default MCPs exist like filesystem (to read/write files in the project), shell (to run commands), etc. We might have used those during dev (for example, to run Docker or git commands via Claude).

- **workflows/**: Possibly we define a workflow file that sequences agents for common tasks. For example, a workflow for "Implement new endpoint" might involve Planner -> writing code -> Tester. Or a workflow for "Update ontology" where Ontology Curator proposes and KG IO applies changes. If we automated the design process, those might be captured here.

- **prompts/** (or fragments): There might be prompt templates used by agents. E.g., a system prompt fragment for all agents to remind them of guidelines (like "Remember this is for a child's AI"). Or specific ones like an emotional style guide for Conversation Coach. These could be stored as separate files for inclusion.

To illustrate, here's an example **subagent YAML** – let's use the **Ontology Curator** (`ontology_curator.yml`) as it's a unique one:

```
name: "Ontology Curator"
role: "Ontology expert AI agent responsible for maintaining and evolving the
OWL ontology and SHACL constraints for the system."
description: |
  - Monitors conversation content and knowledge graph for new concepts or
relationships that are not represented in the current ontology.
  - Proposes additions or changes to the ontology (new classes, properties,
or shape rules) when needed, with justification.
  - Ensures ontology remains in OWL 2 DL and checks consistency using a
reasoner.
  - Validates RDF data against SHACL shapes and flags violations.
tools:
  - type: "http"    # Could use HTTP tool to fetch ontology file or
documentation
  - type: "fuseki"  # Use Fuseki MCP to query/update the triple store
  - type: "shell"   # Possibly to run a reasoner command or OWL validation
script
  - type: "filesystem" # To read/write ontology files (.ttl or .owl) in the
repo
instructions: |
  1. Keep track of the core ontology structure (Child, Conversation,
Utterance, Topic, Emotion, SkillLevel). Before proposing any change, verify
if the concept can be represented with existing classes/properties.

2. If a new class/property is needed (e.g., child mentions an animal and we
have no class for animals), draft an OWL-compatible definition. Use camelCase
for property names and CapitalCase for classes.
  3. Ensure any new addition fits OWL 2 DL profile. Avoid punning or
constructs outside DL.
  4. Update or create SHACL shapes corresponding to new classes/properties.
For example, if adding `hasPet` property, update Child shape to allow that
property if relevant.

5. Validate the ontology after changes using HermiT. If inconsistencies are
found, modify or revert changes.
  6. Output ontology change suggestions in a structured way (e.g., as Turtle
snippets or a summary) so that a human can review.
trigger:
  - event: "New concept encountered"
  - event: "Ontology validation requested"
```

(Explanation: The YAML defines the agent's role and what it should do. The `tools` section lists what MCP servers or actions it can do. In practice, we'd have a mapping so that `type: "fuseki"` corresponds to our configured fuseki MCP that knows how to run SPARQL queries. The instructions provide step-by-step guidance. The trigger events might be used by the Planner to know when to invoke this agent – e.g., after a conversation turn, if new unseen terms are present or periodically to check ontology health.)

Similarly, other agent YAMLs: - **Conversation Coach:** would have instructions about simplifying language to the child's level, adding `**` or `__` around appropriate words, using Spanish variants, etc. It might use no external tools, just operate on text (so no special MCP tools). - **Safety & Tone Guardian:** instructions to cross-check message against disallowed topics or harmful language. It might have access to `mongo` (to read child's forbidden topics list, or maybe we pass that in context so it might not need a tool). - **Memory Retrieval Agent:** would use `qdrant` tool to search, and possibly an embedding model tool (if not done inline). - **Planner Orchestrator:** can use all others. Likely it has logic like: "If user message arrives: call retrieval -> call LLM -> call safety -> call coach -> return result". - **Evaluator/Tester:** might have `shell` access to run tests, `filesystem` to read code, etc. We might not fully implement it, but conceptually it could do things like run `pytest` via shell and report results to us in dev.

MCP Server Config Example (pseudo YAML):

```
server: "fuseki"
base_url: "http://localhost:3030/ds"
endpoints:
  query: "/sparql"   # or /query depending on Fuseki config
  update: "/sparql"  # could be same endpoint or separate /update
auth: none
```

(This would allow agents to send HTTP requests to `http://localhost:3030/ds/sparql` with SPARQL queries or updates. The agent could then get back JSON or XML results.)

Using the Agents in Claude Code: During development, we would engage these subagents as needed: - For instance, after writing the ontology, we could ask the Ontology Curator agent: "Validate the ontology and suggest improvements." The agent would maybe run HermiT via shell, etc., and come back with "No issues found" or suggestions. - While writing code, the Planner agent might automatically spin up the Tester agent to write tests once a feature is done. - We might have a workflow YAML that orchestrates a typical cycle: Plan -> Code (maybe using Claude's main context) -> Test -> Refine.

These .claude files are not needed at runtime, but they document the design logic and ensure that if we use Claude Code to further develop or modify the project, we have a structured approach. We will include them in the repository for transparency and potential re-use.

Conclusion: The `.claude/` directory contains a blueprint of our multi-agent collaboration during development. It captures the specialized roles (subagents) in YAML form and how they interact via MCP. This not only helped us build the system step-by-step but also serves as documentation of the system's intended behavior (each agent's instructions mirror the actual code's function). One could load these into Claude Code to continue evolving the project with the same context.

By providing these YAMLs and prompts, we make the development process reproducible and show how modern AI-assisted coding (with subagents) contributed to our project.

Prompts for PRD.md, TASKS.md, README.md

In order to produce comprehensive documentation with the help of AI (Claude or similar), we prepare specific **prompts** that can be fed to the model to generate the desired documents. Each prompt is crafted to instruct the model on the structure and content needed, ensuring the output is within 500 lines and covers all important points.

Below are the exact prompts we would use for generating **PRD.md**, **TASKS.md**, and **README.md**. These prompts are written in a way that assumes the AI has been primed with context (the architecture, requirements, etc.), which in our case it has from the work we've done.

Prompt for `PRD.md` (Product Requirements Document)

```
**System Name:** Offline Conversational AI Companion for Children with Autism (Spanish/English)

**Overview:**
Provide a brief introduction to the product – an offline-capable AI friend that chats with children (ages 5–13, TEA2 autism) in Spanish. State goals: improve social and communication skills in a safe, controlled manner.

## 1. Goals and Key Features
- **Child-Centric Conversation:** Engage in open-domain chats on topics like school, hobbies, holidays, food, friends. Support Spanish primarily (with some English as needed).
- **Autism-Friendly Design:** Short, clear sentences, consistent tone, and the ability to handle repetitive or literal interpretations. Configurable conversation complexity (levels 1–5) to match each child's ability.
- **Emotion and Tone Markup:** The assistant uses **bold** and __italic__ cues to simulate happy or whispering tones, making emotional context explicit for the child.
- **Safety Controls:** Per-child configurable "do not discuss" topics and strict content filtering to avoid triggers or inappropriate content. The assistant will gently steer away or refuse such topics.
- **Offline Operation:** All AI models (language and speech) run locally without internet, ensuring privacy and availability in offline settings (e.g., therapy centers with no internet).
- **Knowledge Graph Memory:** The system builds a personal knowledge graph of each child's likes, dislikes, and past conversations, allowing it to remember facts (e.g., favorite color) and maintain consistency over long-term interaction.
- **Speech Interface:** Integrate Whisper-based ASR so children can speak and hear (text-to-speech could be future) to make interaction natural for those who prefer voice.

## 2. User Stories
```

- *As a Child (with autism), I want* to talk to my AI friend about my day at school, and have it respond in a friendly, encouraging way that I can understand.
- *As a Parent/Therapist, I want* to configure topics the AI should avoid (like "scary movies") for my child, so I feel secure letting them chat freely.
- *As a Child, I want* the AI to remember things I've told it (like my pet's name) so I feel heard and build a connection.
- *As a Developer, I want* the system to run on a local PC with a GPU, so that no data leaves the device and it works without internet.
- *As an Admin, I want* to browse conversation logs and the knowledge graph via a simple UI to monitor progress and adjust settings.

3. Functional Requirements
Conversation & NLP:
- The assistant shall generate responses in Spanish by default, switching to English only if explicitly configured or if the child uses English.
- It shall adjust the complexity of its language according to the child's **SkillLevel** (1=very simple, 5=near adult). E.g., Level 1: "Eso es bueno." vs Level 5: "Me alegra escuchar que te fue bien en el colegio hoy."
- The assistant's response should include emotional markup in at least 50% of responses, appropriate to context (happy excitement for good news, whisper for reassurance).
- The system should handle at least 5 back-and-forth turns per conversation, maintaining context. It should be able to refer to what the child said earlier in the same session.
- Safety filtering must intercept any response containing blacklisted content (violence, explicit language, personal trauma topics from the profile) and either remove it or replace the response with a gentle refusal/apology.
- The system should avoid introducing sensitive topics on its own. (No unsolicited mentions of potentially upsetting content).
- Topics: The system should recognize and be able to talk about at least the predefined topics (school, hobbies, holidays, food, friends) and be extensible to new ones via configuration (ontology updates for new topics).

Knowledge & Memory:
- A child profile (with name, age (optional), skill level, banned topics) shall be stored and utilized each session.
- The system shall store conversation logs in a database "forever" (unless purged by admin) and be able to retrieve past conversation snippets when relevant to current conversation.
- When a child mentions a new fact about themselves (e.g., "mi color favorito es el azul"), the system should store that in the knowledge graph and recall it later if relevant ("Sé que tu color favorito es el azul.").
- The ontology shall model at minimum the child, conversation sessions, utterances, and topics, plus an "emotion" dimension. It must be possible to run a reasoner to infer new info (not necessarily in real-time, but for data analysis).
- The system shall validate data integrity (via SHACL or otherwise) whenever data is added to the KG to prevent inconsistencies (e.g., skill level out of range, missing required attributes).

```
### Speech:
- The `/asr` API shall transcribe a 5-second audio of the child's speech to
text with at most a 2s delay in *fast* mode, 5s delay in *balanced*, possibly
longer in *accurate*.
- Transcription accuracy in balanced mode should be at least on par with
Whisper large-v2 for Spanish (target ~95% accuracy for clear speech).
- The system should detect the language of speech automatically (Whisper does
this) and transcribe into the same language text. If child says something in
English but profile language is Spanish, it may either translate or respond
indicating it can't (configurable).
- (Future/out-of-scope for v0: text-to-speech to respond orally, not included
yet.)

### Frontend & UX:
- There shall be a simple web-based UI (running locally) for interacting with
the conversation. It should display the chat history and allow input via text
or by recording audio and calling the ASR.
- The UI should visually distinguish the child vs assistant messages, and
render **bold** as a highlighted or bold text, __italic__ as a different
style (maybe grey or italic) to cue the child about tone.
- The UI must have an admin section (password could be added later) where
profiles can be managed (add child, set level, topics to avoid) and where
conversation histories and KG facts for each child can be viewed.
- Bilingual support: UI labels and messages should be available in Spanish
and English (default Spanish). The child's content and assistant's content
are not translated – they appear as produced.
- The UI should be uncluttered, using icons or simple text, suitable for
children with ASD (for example, low-stimulus design: soft colors, clear
fonts, maybe an option to use pictograms for emotions if needed later).

## 4. Non-Functional Requirements
- **Performance:** The system should handle the dialogue in near-real-time
for short texts. Each turn processing (from user message to assistant reply)
ideally < 3 seconds in typical cases (with GPU). Longer utterances or
accurate ASR might push this to ~10s worst-case.
- **Offline & Deployment:** All components run in Docker containers on a
single Ubuntu 22.04 machine with an NVIDIA RTX 5090. No internet connection
is required after installation. Model files are loaded locally. The system
should not attempt to connect externally (for safety and privacy).
- **Scalability:** The design supports only a small number of concurrent
users (practically 1-5) due to the resource-intensive models. This is
acceptable for the use case (usually one child or a small group in a
session). Scaling to many users would require more hardware or model
distillation, which is out of scope.
- **Maintainability:** The code is organized into separate services and
follows clear API contracts. The use of standard databases (Mongo, etc.) and
ontologies means future developers (or researchers) can extend or replace
components (e.g., swap out Qdrant for another vector DB, or update the
ontology) without rewriting the entire system.
- **Security & Privacy:** All data stays on the host machine. Even though we
```

```
skip auth in v0, we assume a trusted environment (the device is controlled by
the caregiver). In future, we'd add at least local network auth if needed.
Data of children is sensitive; by not transmitting it externally, we guard
privacy. We will include a note that the database is not encrypted, so
physical access to the machine equates to data access (potential future
improvement: at-rest encryption).
- **Reliability:** If the LLM model fails to generate a response (e.g., out-
of-memory), the system should handle it by returning a fallback message ("Lo
siento, estoy un poco confundido.") rather than crashing. Similarly, if ASR
fails, the UI can prompt to retry. We aim for graceful degradation.
- **Extensibility:** The ontology-driven approach means we can add new topics
or user attributes just by updating the ontology and maybe adding some
handling in the code. The architecture is modular (each piece can be
replaced/upgraded, e.g., swap Whisper with another ASR model easily, or
replace Qwen with a larger model if hardware allows).

## 5. Project Scope & Deliverables
**In-Scope (MVP v0):**
- Backend APIs: /conv (text in/out), /asr (audio->text), /kg (basic query/
update).
- Basic Spanish conversation ability with level adjustments and safety
filtering.
- Ontology definition covering core concepts; data logging to both Mongo and
Fuseki.
- ASR integration with Whisper large model (running locally).
- Simple frontend to test the conversation (text input/output).
- Dockerfiles for all services and documentation to run them.

**Planned for v1:**
- Polished UI with audio input control and display of emotions (could be via
color or emoji).
- Bilingual interface elements and possibly an English mode for conversation
if needed.
- Enhanced admin tools (e.g., a form to edit ontology suggestions or approve
them).
- More subagent logic in the backend for dynamic adjustments.
- Automated testing and CI setup in repo.

**Out-of-Scope (Future Ideas):**
- Text-to-Speech for assistant responses (so the child hears the answer).
- More complex emotional intelligence (detecting the child's emotion from
voice or text and responding supportively).
- Multi-user support or remote access (our focus is single-machine single-
user for now).
- Gamification or reward mechanisms to encourage engagement.

## 6. Acceptance Criteria
- Deploy the system on a fresh machine with no internet: All containers start
and communicate properly. The assistant can conduct a sample conversation in
Spanish with a test child profile, and obey the configured safety rule (e.g.,
if "ghosts" are banned and child brings it up, assistant safely deflects).
```

```
- The ontology file validates (no OWL DL errors) and SHACL shapes pass for
test data.
- ASR can transcribe a provided sample audio (of a child voice saying a known
phrase) with correct output.
- The UI displays a conversation and the bold/italic markup as intended.
- All unit tests and integration tests are passing, and test coverage is
reasonably high for critical components.
- The Docker images build without error and the total size is manageable
(expected < 10 GB including models). Memory usage fits in 24GB GPU memory for
the models and reasonable CPU/RAM usage.
```

This prompt instructs the AI to produce a structured PRD with goals, user stories, requirements, etc., using the information we have. We've outlined each section needed and some expected content. The AI should output it in Markdown. The PRD.md should be ≤500 lines, which this structure should satisfy.

Prompt for `TASKS.md` (Task Breakdown and Planning)

```
# Project Tasks and Milestones

This document breaks down the implementation tasks for the Offline
Conversational AI Companion project. It is organized by components (Backend,
Frontend, Knowledge Graph, etc.) and includes milestones for v0 (API MVP) and
v1 (UI complete). Each task is tagged with an owner (if applicable),
priority, and an estimate.

## 1. Backend / API
- **Task: Define API Schemas** – Write Pydantic models for /conv request/
response, /asr response, /kg queries. *(Priority: High, Status: Done)*.
- **Task: Implement /conv logic** – Integrate subagent pipeline:
  - Load child profile from Mongo.
  - Query Qdrant for relevant memories.
  - Prompt LLM (Qwen via vLLM) with proper context.
  - Apply safety filter and conversation coach adjustments to LLM output.
  - Save conversation turn to DB and KG (trigger background job).
  *(Priority: High, Owner: Backend Dev, ETA: 5 days)*.
- **Task: Background KG updater** – After each turn, spawn task to extract
triples and insert into Fuseki; also add new vectors to Qdrant.
  *(Priority: High, Owner: Backend Dev, ETA: 3 days)*.
- **Task: /asr endpoint** – Accept audio upload, call Whisper model, return
text. Possibly run in thread to avoid blocking. Ensure it supports `preset`
param logic.
  *(Priority: Medium, Owner: Backend Dev, ETA: 4 days)*.
- **Task: /kg endpoints** – Proxy GET queries and POST updates to Fuseki.
Implement proper parsing of results and error handling.
  *(Priority: Medium, ETA: 2 days)*.
- **Task: Safety Filter Implementation** – Develop a configurable filter
using child's banned topics and possibly a list of bad words. Unit test with
various inputs.
  *(Priority: High, ETA: 2 days)*.
- **Task: Conversation Coach Implementation** – Develop function to adjust
```

```
text: sentence splitting/shortening for low levels, adding ** and __ for
emotions. Use simple heuristic or small classifier for emotion detection (or
based on certain keywords).
  *(Priority: High, ETA: 3 days)*.
- **Task: Integrate vLLM** – Set up vLLM server for Qwen model. Ensure
backend can query it (via openai API compatibility). Test with a simple
prompt.
  *(Priority: High, ETA: 2 days)*.
- **Task: Unit Tests (Backend)** – Write tests for all above: simulate input
to /conv, test filters, etc. Possibly use dummy model responses for
consistency.
  *(Priority: High, ETA: 4 days)*.
- **Task: Integration Test and Tuning** – Run an end-to-end conversation
locally, tune prompt parameters (temperature, etc.) to get desired style.
Adjust as needed.
  *(Priority: Medium, ETA: 2 days)*.

## 2. Knowledge Graph & Ontology
- **Task: Design Ontology (OWL)** – Create initial ontology file with classes
(Child, Conversation, etc.) and properties (hasParticipant, aboutTopic,
etc.). Use Protégé or by hand.
  *(Priority: High, Status: Done)*.
- **Task: Define SHACL Shapes** – Write shapes for data validation
(ChildShape, UtteranceShape, etc. as identified).
  *(Priority: High, ETA: 2 days)*.
- **Task: Set up Fuseki** – Configure a Fuseki Docker that loads the ontology
on startup (could use an assembly or run a post-start script to insert static
ontology triples).
  *(Priority: High, ETA: 1 day)*.
- **Task: KG Sync Mechanism** – Implement the KG I/O agent or equivalent
functions in code to keep Mongo and Fuseki in sync (e.g., after profile
update or new fact extraction).
  *(Priority: High, ETA: 3 days)*.
- **Task: Ontology Curator Logic** – Implement a routine (possibly using LLM)
that monitors if new topics (e.g., words not in ontology) appear. For now,
just log suggestions.
  *(Priority: Low, ETA: 2 days, could be post-v0)*.
- **Task: HermiT Reasoner Integration** – Provide a script or container to
run reasoning. Not needed for MVP runtime, but for validation. Test on
ontology and sample data.
  *(Priority: Medium, ETA: 2 days)*.
- **Task: SPARQL Queries for Memory** – Write and store common queries, e.g.,
get child's interests, check avoid topics. Integrate into backend where
needed (safety checks might use an ASK query).
  *(Priority: Medium, ETA: 1 day)*.

## 3. Frontend (Vue 3 App)
- **Task: UI Framework Setup** – Initialize Vue 3 project, set up bilingual
support (perhaps Vue I18n), basic routing for Chat view and Admin view.
  *(Priority: High, ETA: 1 day)*.
- **Task: Chat Interface** – Create chat component to display conversation.
```

```
Include:
  - Message bubbles (child vs assistant).
  - Bold/italic styling for emotions (CSS for ** and __).
  - Input textbox and send button.
  - Option to record audio and send to /asr (requires integrating mic, maybe
Web Audio API).
  *(Priority: High, Owner: Frontend Dev, ETA: 4 days)*.
- **Task: Admin Interface** – Create views:
  - Child Profile Manager: form to add/edit child (name, level, avoidTopics).
  - Conversation Browser: list past conversations by child and date; click to
view transcript.
  - KG Browser: perhaps a simple list of triples or key-value facts for the
child (like interests, etc.).
  *(Priority: Medium, ETA: 5 days)*.
- **Task: API Integration** – Use Axios or Fetch to call backend:
  - `/conv` on sending a new message (update UI with reply).
  - `/asr` when audio is recorded (show loading indicator while waiting).
  - `/kg` for admin queries (e.g., fetch profile data could also come from
Mongo via a dedicated endpoint, but we might reuse /kg for consistency).
  *(Priority: High, ETA: 3 days)*.
- **Task: Localization** – Ensure all UI text (buttons, titles) is in Spanish
by default, with English translation available. Possibly a toggle in admin to
switch language.
  *(Priority: Low, ETA: 2 days)*.
- **Task: Frontend Testing** – Manually test UI flows:
  - Send message, get response (appearance of bold/italic).
  - Record short audio, see it transcribed and get response.
  - Edit profile (e.g., add avoid topic "monsters"), then attempt that
conversation to see if assistant avoids it.
  *(Priority: Medium, ETA: 2 days)*.
- **Task: Styling and UX Polish** – Make sure the interface is child-
friendly: large font option, simple colors, maybe an avatar icon for the
assistant (friendly graphic). Keep layout responsive (if they use a tablet).
  *(Priority: Low, ETA: 3 days, ongoing tweaking)*.

## 4. DevOps / Deployment
- **Task: Write Dockerfiles** – One for each: backend API, ASR service,
frontend, reasoner job. Multi-stage builds as per design.
  *(Priority: High, ETA: 2 days, can parallelize)*.
- **Task: Docker Build & Run Scripts** – Possibly create a simple shell
script or instructions to run all containers (since no compose). Test on a
fresh machine.
  *(Priority: High, ETA: 1 day)*.
- **Task: Model Downloads** – Document or automate the download of Qwen-3
model and Whisper model. Possibly add a step in Docker build or entrypoint to
check for model files.
  *(Priority: High, ETA: 1 day)*.
- **Task: Pre-commit & CI Setup** – Configure pre-commit hooks (ruff, black,
etc.) and a GitHub Actions (or local CI pipeline) to run tests on push.
Ensure a GPU runner is available for ASR/LLM tests or mark them optional.
  *(Priority: Medium, ETA: 2 days)*.
```

```
- **Task: Monitoring/Logging** – Ensure that each service logs important
events. For example, the backend should log whenever a safety filter triggers
or a KG update happens (for debugging). Possibly integrate a simple logging
library with timestamps. Not a separate service, just an implementation
detail, but worth tracking as a task.
  *(Priority: Low, ETA: 1 day)*.

## 5. Milestones
- **Milestone v0: Core API Complete (ETA: Month 1)**
  - Criteria: Able to have a text-based conversation via `/conv` with safety
and tone adjustments working. Ontology in place and data being stored (even
if not heavily utilized yet). ASR working via API (maybe not integrated into
UI yet). Basic test coverage. No frontend or just a rudimentary one.
- **Milestone v1: Full System with UI (ETA: Month 2)**
  - Criteria: User can use the Vue app to talk (text or voice) with the
assistant. Admin can configure profile and see knowledge graph info. All
containers running together offline. Refinements to conversation style after
testing with target users.
- **Milestone v2: Polishing and Extensions (ETA: Month 3)** – (Future) Add
nice-to-have features like TTS, further content tuning, multi-child support,
etc., based on feedback.

## 6. Risks and Mitigations
- *Model performance:* Qwen-3 8B may be borderline for quality. If responses
are not sufficient, consider switching to 30B (if hardware allows) or using
online Claude for planning with local execution. Mitigation: provide an easy
way to plug in a different model.
- *ASR misrecognition:* Could lead to misunderstandings. Mitigation: at level
1–2, maybe confirm what the child said ("Creo que dijiste que… ¿es
correcto?").
- *Ontology complexity:* Over-engineering KG could eat time without immediate
benefit. Mitigation: focus on a few key uses (like tracking interests) in
early versions.
- *Timeline risks:* Integration of many components could cause delays. To
mitigate, we have parallel tracks (e.g., frontend and backend dev can proceed
largely independently with a mocked API).
```

This TASKS.md prompt guides the AI to enumerate tasks in a structured way, including priorities and estimates. We made sure it's comprehensive, covering everything from development tasks to milestones and risks.

Prompt for `README.md` (Project README and Setup Guide)

```
# Offline AI Friend for Kids (Autism Support) – README

Welcome to the Offline AI Friend project! This repository contains a
Dockerized application that lets a child with autism have conversations with
an AI assistant completely offline. The assistant speaks Spanish (and
English) and adapts to the child's communication level and preferences.
```

````
## Project Structure
- **backend/** – FastAPI app that serves:
  - `/conv` – conversation endpoint (child message in, assistant reply out).
  - `/asr` – speech-to-text endpoint using Whisper.
  - `/kg`  – knowledge graph query/update endpoint.
  It contains submodules for safety filters, ontology sync, etc.
- **frontend/** – Vue 3 application (in `frontend/`) that provides a web UI for chatting and administration.
- **ontology/** – OWL ontology files and SHACL shapes defining the knowledge graph schema.
- **.claude/** – Configuration for Claude Code subagents (used during development for planning, optional).
- **docker/** – Dockerfiles and any helper scripts for building images.
- Other files: `PRD.md` (product requirements), `TASKS.md` (task planning), etc.

## Quick Start (Running the System)
These steps assume you have **Docker** and **docker-compose** installed (if using compose for convenience; otherwise, manual commands are provided).

1. **Clone the repo**:
   ```bash
   git clone https://github.com/your-org/offline-ai-friend.git
   cd offline-ai-friend
   ```

2. **Obtain Models**: Due to size, the large language model and ASR model are not packaged in the repo.
   - Download the Qwen-3 8B model weights:
     - Option 1: Use `vllm` to download on first run (it will fetch from HuggingFace).
     - Option 2: Manually download `Qwen/Qwen3-8B` from HuggingFace and place it in `./models/qwen3-8b/`.
   - Download or prepare Whisper large-v3 model for faster-whisper:
     - Easiest: run `python -m faster_whisper download-model large-v2` (large-v3 uses same architecture). Or use the provided conversion script in `asr/`.
     - Place the model in `./models/whisper-large-v3/` (if not, the ASR container will attempt to download on startup, which requires internet).
   - Ensure these directories and files are accessible to Docker (if you use volume mounting as below).

3. **Build Docker images**:
   We provide a `docker-compose.yml` for convenience:
   ```bash
   docker-compose build
   ```
   This will build:
   - `backend` image (FastAPI + our code),
   - `frontend` image (Vue app),
   - `asr` image (Whisper model and API),
````

```
   - also pulls `mongo:6`, `qdrant` and `fuseki` images.
   *(If not using compose, see manual build commands in docs.)*

4. **Run the containers**:
   ```bash
   docker-compose up -d
   ```
   This starts everything in background. Give it a bit of time on first run (the ASR service might optimize the model on first load).

   If doing manually:
   ```bash
   # start databases
   docker network create ai-friend-net
   docker run -d --network ai-friend-net --name mongo mongo:6.0
   docker run -d --network ai-friend-net --name qdrant -p 6333:6333 qdrant/qdrant:v1.3.0
   docker run -d --network ai-friend-net --name fuseki -p 3030:3030 ghcr.io/stain/jena-fuseki:latest
   # load ontology into fuseki (optional step: you can do via UI at http://localhost:3030 or a curl command).
   docker run -d --network ai-friend-net --name asr -p 5000:5000 offline-asr:latest
   docker run -d --network ai-friend-net --name backend -p 8000:8000 offline-backend:latest
   docker run -d --network ai-friend-net --name frontend -p 81:81 offline-frontend:latest
   ```
   (Using a network ensures containers can talk by name; backend expects services named "mongo", "qdrant", etc.)

5. **Access the application**:
   - Frontend UI: Open `http://localhost:81` in your browser. You should see the chat interface.
   - If you prefer using the APIs directly (e.g., for testing):
     - `http://localhost:8000/docs` – interactive Swagger docs for the FastAPI endpoints.
     - You can POST to `http://localhost:8000/conv` with JSON to get a response.

6. **Try a conversation**:
   - In the UI, select (or create) a child profile (if none, it might prompt to create one).
   - Type a message like “Hola, ¿cómo estás?” and send. The assistant should respond in Spanish, possibly with some **markup** for emotion.
   - Use the microphone button (if enabled) to record a short sentence. Upon stop, it will call ASR and the text will appear, then the assistant replies.
   - Check that if you mention something the profile disallows, the assistant responds evasively or changes subject.

7. **Admin features**:
```

```
   - On the UI, navigate to “Profiles” or similar section. There you can edit the child’s preferences (e.g., add “monstruos” to avoided topics).
   - Navigate to “Conversations” to see saved logs. And “Knowledge Graph” to see facts (like the ontology or extracted info). These sections are rudimentary in v0.

## Configuration
- **Child Profiles:** stored in MongoDB. You can edit them via UI or directly in the DB (not recommended). The structure includes name, skillLevel, avoidTopics, etc.
- **Safety Words:** We have a file `backend/safety_rules.yaml` (for example) that contains global forbidden words or phrases. This is loaded at startup. You can customize it (e.g., add profanity words etc.).
- **Environment Variables:**
  - `MODEL_PATH` for Qwen model (if using local file) – point it to the directory or file for the model.
  - `WHISPER_MODEL_PATH` for the ASR model directory.
  - `FUSEKI_DATASET` if your fuseki dataset is named differently (default is `/ds`).
  These can be set in the docker-compose.yml or Docker env.

- **Language Settings:** By default, assistant will respond in the profile’s language (currently always Spanish unless changed). If you want to force English responses, set the child’s preferred_language in profile or tweak the system prompt.

## Development
If you want to develop or extend this project:
- Install Python 3.11 and Node 20 on your machine.
- Backend: `pip install -r requirements.txt`. It uses FastAPI, pydantic, etc. Run `uvicorn main:app` for dev server.
- Frontend: `npm install`, then `npm run dev` for hot-reload.
- You might need a local Mongo and Qdrant running. Qdrant can be started via Docker easily, same for Mongo.
- Ontology editing can be done with Protégé (open `ontology/ontology.owl`).
- We used Anthropic’s Claude with subagents for planning; you can see `.claude/agents` for how tasks were structured (not needed to run the app, but interesting for understanding design).

## Testing
- Run `pytest` to execute the test suite. Some tests (like ASR) might be skipped if models not present or if no GPU. You can run them specifically by marking with `pytest -m asr`.
- We also include a Postman collection (if applicable) or you can use the `/docs` UI to manually test endpoints.
- Linting: `ruff .` and `black .` should report no issues (enforced via pre-commit). Mypy type checks should pass (some third-party libs might be excluded).

## Known Issues and Limitations
- **Response Quality:** Using an 8B model means sometimes responses are
```

```
simplistic or may go off-script. We mitigate with tight prompts and
retrieval, but it's not perfect. For more fluent conversations, a larger
model or online model would be needed.
- **Single-user assumption:** The system isn't secured for multi-user. If you
connect multiple clients to the API, they'd share the same DB and could
interfere (no user auth tokens, etc.). In a controlled setting, this is fine.
- **ASR Performance:** On CPU or without AVX support, transcription will be
slow. GPU is recommended. Also, Whisper might not perfectly transcribe child
speech, especially if pronunciation is atypical; we did not fine-tune on
child speech data.
- **Ontology Use:** As of v1, the ontology is not deeply integrated into
generating answers (we use it more for consistency checks and memory). In
future, the assistant might query the KG to answer factual questions the
child asks about themselves ("Do I like X?").
- **Docker Memory:** Make sure to allocate enough memory to Docker for the
model loads (at least 16GB RAM and the GPU memory for models). If the backend
container dies on start, it might be OOM when loading Qwen – in that case,
try the 4B model or ensure you have swap.

## References
This project references a number of open-source tools and research:
- **Anthropic Claude** – for planning the architecture (some planning docs
included).
- **Model Context Protocol (MCP)** – used conceptually for tool use [27].
- **Alibaba Qwen-3 model** – small LLM used for conversation [4].
- **OpenAI Whisper** – speech recognition model (large-v3) via faster-whisper
for speed [11].
- **Apache Jena Fuseki** – RDF triple store for knowledge graph [7].
- **SHACL** – for validating RDF data integrity [15].
- **HermiT Reasoner** – to ensure ontology consistency and infer new
knowledge [16].
- And others (see `References.md` for more).

Feel free to explore those links to understand the underpinnings.

## Contact and Acknowledgments
This project was developed by [Your Team] in 2025. We thank the open-source
community for the tools that made it possible. If you have questions or want
to contribute, please open an issue or contact us at [email].
```

This README prompt ensures the AI produces a comprehensive guide with setup instructions, usage, and important notes, suitable for developers and users.

```
With these prompts prepared, we would feed them to Claude (or another LLM) to
generate the corresponding documents. Each prompt is structured to remind the
model of the necessary content and format (using Markdown headings, lists,
code blocks, etc. as appropriate). The model's output would then be reviewed
and edited as needed before finalizing the docs.
```

We've provided the prompts above in Markdown format enclosed in triple backticks for clarity (as if they are files in the repo). These are ready to run in the sense that a user can copy them into a Claude session that has been primed with project context, and get nicely formatted documentation as output.

v0/v1 Acceptance Checklist

Finally, to ensure we meet our goals, here are checklists for v0 (minimal viable product without full UI) and v1 (first complete version with UI and extras):

v0 (API-Only MVP) Checklist:

- [ ] **Backend**: /conv returns a well-formed JSON response given a sample input. The assistant's reply is generated (even if simple) and contains correct fields and markup.
- [ ] **Basic Spanish Response**: Confirm the assistant responds in Spanish and uses short sentences for a low skill level input.
- [ ] **Safety Filter**: Test by sending a disallowed topic in the message (as per a test profile) – verify the assistant's reply does not contain that topic and perhaps gives a safe refusal. No unsafe content passes through.
- [ ] **ASR**: /asr can transcribe an audio sample to text. Try with a known Spanish phrase audio – result text matches expected phrase (or is reasonably close).
- [ ] **Data Stores Running**: Mongo, Qdrant, Fuseki containers all up and accessible. After a conversation turn, verify:
- The conversation and utterance are saved in Mongo (e.g., conversation document has the new turn).
- The triple store has new data (e.g., a new Utterance individual, or child's stated fact appears).
- Qdrant has an increased count of vectors.
- [ ] **Ontology Consistency**: Run the reasoner on the ontology + instance data (we might do this manually with our tool) – it should report no inconsistencies. All SHACL shapes pass (we can run a validation and ensure no errors for the test data).
- [ ] **Endpoints Test**: Using the OpenAPI docs or curl, test a simple flow:
- call /conv with "Hola" -> get reply.
- call /kg to query the child's name or skill level -> correct result. Ensure no 5xx errors and responses conform to schema.
- [ ] **Docker Build & Run**: Able to build the images and run all necessary containers on a test machine with no outside network. The system should function offline (if we simulate offline by blocking internet, it still works using local models).
- [ ] **Logs and Errors**: Check backend logs for any obvious errors or tracebacks. v0 should run without crashing or spamming errors in logs during normal operation.
- [ ] **Documentation**: PRD, TASKS, README generated and reviewed for accuracy. (Though not a code criteria, it's deliverable completeness.)

If all above are checked, v0 can be considered achieved.

v1 (Full System with UI) Checklist:

- [ ] **All v0 items** – Regression check that all v0 functionality still works in v1.
- [ ] **Frontend Chat**: A user can open the web UI, select a profile (or create one), and exchange at least 3 messages with the assistant. The messages appear in a chat bubble format, the assistant's bold/italic markup is rendered correctly (e.g., bold text appears bold). No page reload needed; it updates dynamically.

- [ ] **Voice Input**: On the UI, the record button triggers audio capture and sending to /asr. The transcribed text appears (either directly in chat or pre-filled in input). The assistant then responds. This loop should work at least 1 time in a test. (Timing may vary, but it should eventually respond.)
- [ ] **Profile Management UI**: Able to view a list of children, add a new child (with name and level), edit an existing child's preferences (e.g., add an avoided topic, change level). Verify the changes reflect: e.g., after changing level from 5 to 1, the assistant's next response becomes simpler.
- [ ] **Data Browsing UI**: On an admin page, conversation logs are visible. Clicking a conversation shows the turns. Knowledge graph info is accessible (could be as simple as listing triples like "Alice hasInterest chocolate" etc.). Ensure this doesn't show technical IDs unintelligible to users (maybe we present readable labels).
- [ ] **Bilingual Toggle**: If implemented, switching the UI language (say, to English) changes the interface text. (The assistant's conversation language with child remains Spanish unless profile changed – that's separate.)
- [ ] **Ontology Suggestions Workflow**: Induce an ontology suggestion scenario (e.g., child talks about "dinosaur" when no such class exists). Check backend logs or DB that an ontology suggestion was recorded (like a log entry "Suggest adding Topic dinosaur"). In the UI, if we have a section for suggestions, ensure it appears. Accept the suggestion (even if it's just manually editing ontology for now) and verify that it's reflected (e.g., "dinosaur" now in allowed topics).
- [ ] **Performance**: Using a typical setup (RTX 5090, etc.), the end-to-end response time for a short text input is reasonable (<3s for most). For voice input of ~5s, transcription <5s and response <3s, total ~8s or so. (We can tick if subjective performance is okay).
- [ ] **Stability**: Run a longer conversation (say 10 turns). The system should not slow to a crawl or crash. Memory usage stable (no major memory leak).
- [ ] **Docker**: All containers for v1 (including front-end) work together via compose or manual. The front-end can talk to back-end through the network (if not on same host network, ensure CORS or correct hostnames are set). No console errors in browser indicating API call failures.
- [ ] **Testing Updated**: All automated tests updated for new UI/API changes (if any endpoints changed). They pass. We added perhaps some integration tests for the frontend (maybe not fully automated, but at least manual test plans).
- [ ] **Documentation Updated**: README now includes front-end usage, any new env variables, etc. Also perhaps a user guide in docs if needed. The PRD and design docs reflect any changes we made during v1 (if any requirement adjustments).

When all v1 items are checked off, we have a fully functional system ready for a user acceptance test in a real scenario.

By following these checklists, we can systematically verify that both the MVP and the full release meet the agreed requirements. This reduces the chance of missing something important (like a safety filter not actually active, or the UI not properly wired to the back-end). Each item is phrased as an observable verification.

This completes the deep research and planning task. We have provided detailed architecture, design decisions, specs, and supporting references.

References (with links)

- Anthropic, *Introducing the Model Context Protocol (MCP)* – open standard for connecting AI to external systems [27] [28].

- Alibaba Qwen Team, *Qwen3-VL 4B/8B Release Notes*, MarkTechPost, Oct 2025 – small Qwen3 models (4B/8B) retain full capabilities with FP8, recommended to serve via vLLM [4] [2].
- vLLM Documentation – using vLLM to host models like Qwen3-8B on local OpenAI-like API [1].
- SYSTRAN, *faster-whisper* GitHub – Whisper reimplementation up to 4× faster than original, with same accuracy [11].
- HuggingFace, *Whisper large-v3 Model Card* – details on Whisper v3 training and 10-20% error reduction vs v2 [29].
- Apache Jena, *Fuseki Server Documentation* – default SPARQL port is 3030 (–port option default 3030) [7] [8].
- OpenAI Cookbook, *Using Qdrant vector database* – Qdrant is an open-source vector DB for embeddings, supports filtering with payloads [10].
- ResearchGate, *HermiT OWL Reasoner* – HermiT supports all OWL 2 DL axioms and is optimized for performance on large ontologies [16].
- Wikipedia, *SHACL (Shapes Constraint Language)* – SHACL is a W3C standard to define constraints on RDF graphs (cardinalities, data types, etc.) [15].
- Wikipedia, *Web Ontology Language (OWL 2)* – OWL 2 is the 2009 version of the ontology language, supported by reasoners like Pellet and HermiT [17].
- Docker Docs, *Multi-stage builds* – technique to produce tiny production images by copying only needed artifacts to final stage [24].

---

[1] [5] [6] vLLM - Qwen
https://qwen.readthedocs.io/en/latest/deployment/vllm.html

[2] [3] [4] Alibaba's Qwen AI Releases Compact Dense Qwen3-VL 4B/8B (Instruct & Thinking) With FP8 Checkpoints - MarkTechPost
https://www.marktechpost.com/2025/10/14/alibabas-qwen-ai-releases-compact-dense-qwen3-vl-4b-8b-instruct-thinking-with-fp8-checkpoints/

[7] [8] Apache Jena - Fuseki: serving RDF data over HTTP
https://jena.apache.org/documentation/archive/serving_data/fuseki1.html

[9] Local Quickstart - Qdrant
https://qdrant.tech/documentation/quickstart/

[10] Using Qdrant as a vector database for OpenAI embeddings
https://cookbook.openai.com/examples/vector_databases/qdrant/getting_started_with_qdrant_and_openai

[11] [19] [20] [21] [22] [23] [25] GitHub - SYSTRAN/faster-whisper: Faster Whisper transcription with CTranslate2
https://github.com/SYSTRAN/faster-whisper

[12] [13] Whisper Large V3 Turbo: As Good as Large V2 but 6x Faster | Medium
https://medium.com/@bnjmn_marie/whisper-large-v3-turbo-as-good-as-large-v2-but-6x-faster-97f0803fa933

[14] [18] [29] openai/whisper-large-v3 · Hugging Face
https://huggingface.co/openai/whisper-large-v3

[15] SHACL - Wikipedia
https://en.wikipedia.org/wiki/SHACL

[16] HermiT: A highly-efficient OWL reasoner - ResearchGate
https://www.researchgate.net/publication/221218516_HermiT_A_highly-efficient_OWL_reasoner

[17] Web Ontology Language - Wikipedia
https://en.wikipedia.org/wiki/Web_Ontology_Language

24 26 Multi-stage | Docker Docs
https://docs.docker.com/build/building/multi-stage/

27 28 Introducing the Model Context Protocol \ Anthropic
https://www.anthropic.com/news/model-context-protocol