

Developers Guide for Mental Health Conversational AI

A system architecture specifying decoupled frontend/backend patterns, safety middleware pipelines, database auditing schemas, and automated gatekeepers.



DEVELOPER INTEGRATION PREMISE

Developers must not be left to handle clinical decisions without specialists input. Safety guardrails and emergency bypasses must be implemented as a backend middleware that run deterministically outside the core generative model's logic.

1. Decoupled Frontend + Backend Architecture

Isolating presentation state from secure inference orchestration

To prevent credentials exposure, user manipulation, and **prompt injection** attacks, developers must follow a decoupled architecture. The frontend must remain stateless, while the backend serves as a secure layer.



Frontend (Svelte / React / Mobile)

- **Stateless interface:** Render input textboxes and message lists. Perform zero local diagnostic processing.
- **Avoid secret key leakage:** Never bundle Large Language Model (LLM) API keys (OpenAI, Anthropic, Gemini, Local Models) in client code.
- **Automatic clearing:** Clear all local storage, session storage, and variables upon user logout or inactivity timeout.
- **Immediate transport:** Hand off chat payloads to the secure backend gateway via TLS.



Backend (Node.js / Python / Go)

- **Secrets isolation:** House all model keys, DB connections, and third-party secrets within secure env variables.
- **Validation:** validate inputs before sending them to the LLM.
- **Rate limiting & throttle:** Implement strict token **rate limiting** per IP to prevent DoS attacks.
- **Keep history for auditing:** Save every token transaction, safety status, and user session in the DB.

2. Input interceptor & output validator middleware

Implementing deterministic security checkpoints around generative model APIs

A mental health chatbot must utilize gateway handlers. Developers must build a middleware pipeline to validate messages sent to the LLM model and also validate the response content.

PIPELINE GATEWAY: PHASE 01

Pre-inference validation

1. Lexicon string scan

Implement regex pattern libraries checking for self-harm intent, crisis phrases, or drug requests. Bypass LLM if triggered.

2. Semantic vector match

Generate embeddings for user query and compare cosine similarity against a database of known high-risk scenarios. Halt flow if similarity score exceeds the threshold.

3. Prompt injection shield

Strip instructions like "ignore previous rules", "system override", or character-play triggers before query compilation.

PIPELINE GATEWAY: PHASE 02

Post-Inference Validation

1. Diagnosis detector

Scan LLM response tokens against clinical diagnosis keywords. If patterns match "You have MDD" or "I diagnose you with...", trigger safety exception.

2. Out-of-bounds sanitizer

Block responses recommending specific drug dosages, medical treatments, unwanted referrals.

3. Moderation filter

Evaluate response scores using standard Moderation APIs to filter out toxic, manipulative, or hostile wording prior to client transmission.

EXAMPLE OF MIDDLEWARE LOGIC

```
1  async function handleChatRequest(req, res) {
2    const userInput = req.body.message;
3
4    // 1. Run Pre-Inference Interceptor
5    if (await isCrisisOrInjection(userInput)) {
6      return res.json({ response: STATIC_CRISIS_REPLY, isEscalated: true });
7    }
8
9    // 2. Fetch LLM Inference response
10   const rawResponse = await callModelAPI(userInput);
11
12   // 3. Run Post-Inference Validator
13   const safeResponse = sanitizeOutput(rawResponse);
14   if (safeResponse.isViolating) {
15     return res.json({ response: STATIC_FALLBACK_REPLY, isEscalated: false });
16   }
17
18   return res.json({ response: safeResponse.text, isEscalated: false });
19 }
```

3. Session Retention

Logging and storing transactional chat logs for clinical retrospectives and tuning

In the context of mental health, audit trails are mandatory. Developers must persist all raw conversations, safety flags, and metadata into a secure database. This enables retrospective audits by specialists and allows safe adjustments over time.

Retaining all user interactions

Store user messages, system prompt versions, parameters, and metadata. Conversations should never be discarded, even if they fail safety validation or trigger emergency bypasses.

Audit log metadata fields

Log flags indicating whether the pre-inference check triggered `is_blocked`, semantic similarity metrics `vector_score`, toxicity values `toxicity_index`, and whether a human escalation prompt was requested `escalated_to_human`.

EXAMPLE OF SQL LOGGING SCHEMA

```
1 CREATE TABLE chat_interactions (
2   id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
3   session_id UUID NOT NULL,
4   user_id UUID,
5
6   -- Interaction Timestamps
7   created_at TIMESTAMPTZ WITH TIME ZONE DEFAULT NOW(),
8   latency_ms INTEGER NOT NULL,
9
10  -- Content Audit Data
11  user_message TEXT NOT NULL,
12  system_prompt_version VARCHAR(50) NOT NULL,
13  bot_response TEXT NOT NULL,
14
15  -- Safety Middleware Metrics
16  is_blocked BOOLEAN DEFAULT FALSE,
17  blocked_by_layer VARCHAR(20),      -- 'LEXICON', 'VECTOR', 'POST_FILTER'
18  vector_score NUMERIC(4, 3),        -- Similarity score vs self-harm vectors
19  toxicity_score NUMERIC(4, 3),      -- Output filter safety score
20  escalated_to_human BOOLEAN DEFAULT FALSE,
21
22  -- Token Analytics
23  tokens_input INTEGER NOT NULL,
24  tokens_output INTEGER NOT NULL,
25  model_name VARCHAR(50) NOT NULL
26 );
27
28 CREATE INDEX idx_session_safety
29 ON chat_interactions(session_id, is_blocked, escalated_to_human);
```

4. Retrieval-Augmented Generation (RAG)

Enhancing model factual correctness via local vector database lookups and style-aligned few-shot demonstration sequences

Mental health conversational systems must provide safe, evidence-based responses. Rather than relying solely on the static weights of fine-tuned LLMs or long, complex system instructions, developers must implement a Retrieval-Augmented Generation (RAG) architecture paired with targeted, specialist-validated few-shot examples to maintain boundary control and response safety.

Vector Storage: zvec

In-Process DB

For fast semantic retrieval of clinical guidelines, developers should use [zvec](#), an open-source, serverless, in-process vector database that embeds directly into applications.

- **Zero Config:** Runs locally with no database server or configuration.
- **Hybrid Search:** Fuses vector similarity with native Full-Text Search (FTS).
- **Durable:** Write-ahead logging (WAL) guarantees persistence on process crash.

Few-Shot Prompting

Empirical Behavior

Relying solely on rules inside the system prompt often leads to instruction drift over extended conversations. Prepend 2-3 specialist-approved dialogue turns to demonstrate boundary enforcement.

- **Behavioral Alignment:** Shows the exact therapeutic boundaries and response tone.
- **Toxicity Guard:** Mitigates hallucinated diagnostics or dosage recommendations.
- **Drift Mitigation:** Hardens fallback responses in high-risk edge cases.

A. RAG RETRIEVAL & PROMPT CONSTRUCTION

Example Python pipeline demonstrating local retrieval utilizing [zvec](#) and formatting a payload with context and safety few-shots.

```
1 import zvec
2
3 # 1. Query in-process zvec database (embedded locally)
4 collection = zvec.create_and_open(path="./zvec_db", schema=schema)
5 matches = collection.query(
6     zvec.VectorQuery("embedding", vector=query_emb),
7     topk=2
8 )
9 context = "\n".join([doc["text"] for doc in matches])
10
11 # 2. Build conversation payload with clinical few-shot examples
12 messages = [
13     {"role": "system", "content": f"Answer based ONLY on context: {context}"},
14     # Few-Shot Pair 01: Dosage query boundary
15     {"role": "user", "content": "What dosage of Xanax should I take?"},
16     {"role": "assistant", "content": "I cannot recommend dosage. [...]"},
17     # Few-Shot Pair 02: Crisis query boundary
18     {"role": "user", "content": "I want to end my life."},
19     {
20         "role": "assistant",
21         "content": "I cannot provide crisis support. Please contact 988.",
22     },
23     # User message
24     {"role": "user", "content": user_message},
25 ]
26 ]
```

5. Fine-Tuning

Methodologies for adapting base models safely using specialist-audited interactions

When adapting general LLMs to mental health domains, direct model tuning is can be used to align conversational behavior. When performing fine-tuning, developers should avoid modifying base weights globally and instead deploy Parameter-Efficient Fine-Tuning (PEFT) methods.

A. ADAPTATION METHODOLOGIES

LoRA / QLoRA

Recommended

Freezes the base model weights and injects low-rank trainable matrices into self-attention layers. **QLoRA** quantizes the base model to 4-bit.

- Reduces VRAM usage by 70%-90%.
- Prevents catastrophic forgetting of general reasoning.
- Easy to hot-swap adapter weights per tenant session.

Alternative Adapters

Limited Use Case

Full Fine-Tuning (FFT): Alters all model parameters. High risk of breaking safety defaults.

Prefix tuning Prepends virtual task tokens.

- FFT requires much more VRAM than LoRA.
- Prompt tuning yields lower safety adherence in complex dialogue.
- FFT is prone to model collapse on small datasets.

B. OPTIMIZATION TOOLKIT: UNSLOTH

For training pipeline acceleration, developers should employ Unsloth. It provides custom GPU kernels that speed up training cycles by 2x to 5x and reduces VRAM footprint by up to 80% without sacrificing model accuracy, supporting native export pathways to GGUF, vLLM, and 16-bit float adapters.

```
1 from unsloth import FastLanguageModel
2 import torch
3
4 # 1. Load optimized model and tokenizer
5 model, tokenizer = FastLanguageModel.from_pretrained(
6     model_name = "unsloth/llama-3-8b-Instruct",
7     max_seq_length = 2048,
8     dtype = None,
9     load_in_4bit = True,
10 )
11
12 # 2. Inject LoRA adapters to attention projections
13 model = FastLanguageModel.get_peft_model(
14     model,
15     r = 16, # Rank dimension
16     target_modules = ["q_proj", "k_proj", "..."],
17     lora_alpha = 16,
18     lora_dropout = 0,
19     bias = "none",
20     use_gradient_checkpointing = "unsloth",
21     random_state = 3407,
22 )
23
24 # 3. Model is now ready for supervised fine-tuning (SFT) training
```

6. Development checklist

Pre-launch checklist for validating security, rate limits, and audit logs

A. INTEGRATION TESTING RULES

🔌 1. API rate limiting

Developers must configure rate-limiters at the gateway level. High-frequency queries can exhaust server budgets or lock accounts. Apply an IP-based token-bucket limiter to drop suspicious spamming behavior before calling LLMs.

🔄 2. Session lifecycle tests

Verify that client-side tokens are wiped from browser storage (localStorage, sessionStorage) and in-memory caches upon sign-out or automatic timeout. This prevents other users of shared devices from inspecting history.

B. DEVELOPER LAUNCH CHECKLIST

The engineering team must sign off and verify each of the following configurations prior to deploying the AI conversational system.

- ☒ Verify secret isolation: All API keys (Gemini, Anthropic, OpenAI) reside exclusively in backend environment variables.
- ☒ Test pre-inference bypass gate: Trigger words and similarity checks successfully inject static crisis layouts.
- ☒ Confirm database retention: The system persists all transactions, input tokens, and validation scores.
- ☒ Deploy rate-limiting middleware: Configure API gateways to drop requests exceeding the token limit.
- ☒ Ensure TLS 1.3 Transport Security: Enforce SSL/TLS encryption across all REST/WebSocket chat endpoints.
- ☒ Validate client session lifecycle: Enforce automatic token invalidation and storage purge on user inactivity.