
Spur – Founding Full-Stack Engineer Take-Home

Deadline: 31st December 2025

Context

Spur is a “boring makes money” customer engagement & automation platform.
We power:

- AI agents on WhatsApp, Instagram, live chat & Facebook
- WhatsApp bulk messaging & automation
- Integrations with Shopify, Zoho, Stripe, Razorpay, LeadSquared, etc.

For this assignment, you’ll build a **mini AI support agent for a live chat widget**. This is very close to what you’d work on as a founding engineer at Spur.

Tech Stack (Guidelines, not hard rules)

We’d *prefer* you use some/all of:

- **Backend:** Node.js + **TypeScript**
- **Frontend: Svelte** (or SvelteKit). React/Vue/etc. are okay if you’re faster there.
- **Database: PostgreSQL** (or a simple SQL DB; SQLite is fine for the exercise)
- **Cache:** Redis (optional, nice-to-have)

Use whatever lets you move quickly *and* write clean code. Don’t integrate Shopify / Facebook / Instagram / WhatsApp APIs for this – just the LLM integration.

The Assignment: AI Live Chat Agent

Goal

Build a small web app that simulates a **customer support chat** where an **AI agent answers user questions** using a real LLM API (OpenAI / Claude / etc.).

Core User Flow

1. User opens a web page with a chat widget/panel.

2. User types a message (“What’s your return policy?”, “Do you ship to USA?”, etc.).
3. Frontend sends the message to your backend.
4. Backend:
    - Logs/persists the conversation.
    - Calls an **LLM API** with some prompt/context.
    - Returns the LLM’s reply.
5. Frontend displays the AI agent’s answer in the chat UI.

Functional Requirements

1. Chat UI (Frontend)

- Simple live chat interface:
    - Scrollable message list.
    - Clear distinction between **user** and **AI** messages.
    - Input box + send button (Enter should also send).
- Auto-scroll to latest message.
- Basic UX niceties (examples, not mandatory):
    - Disabled send button while request in flight.
    - Optional: “Agent is typing…” indicator.

2. Backend API

- Implement a backend server in **TypeScript**.
- Expose at least:
    - `POST /chat/message` – accepts `{ message: string, (optional) sessionId: string }`
    - Returns `{ reply: string, sessionId: string }`
- The backend should:
    - Persist every message (user + AI) to a database.
    - Associate messages with a session/conversation.
    - Call a **real LLM API** to generate the reply.

3. LLM Integration (Required)

- Integrate with **any** major LLM provider (e.g. OpenAI, Anthropic / Claude, etc.).
- Use an **API key via environment variables** (don’t commit secrets).
- Wrap the LLM call behind a function/service, e.g. `generateReply(history, userMessage)`.
- Prompt design is up to you, but do something simple like:

    - System prompt: “You are a helpful support agent for a small e-commerce store. Answer clearly and concisely.”
    - Include some conversation history so replies are contextual.
- Add basic **guardrails**:
    - Handle LLM/API errors (timeouts, invalid key, rate limit) gracefully and return a friendly error message to the user.
    - Optionally cap max tokens / messages for cost control (document any assumptions).

4. FAQ / Domain Knowledge

- Seed the agent with some basic “knowledge” about a fictional store, e.g.:
    - Shipping policy
    - Return/refund policy
    - Support hours
- You can:
    - Hardcode this in your prompt, **or**
    - Store it in the DB and include it in the prompt
- The AI should be able to answer these FAQs reliably.

5. Data Model & Persistence

- Persist at least:
    - `conversations` (id, createdAt, maybe metadata)
    - `messages` (id, conversationId, sender: "user" | "ai", text, timestamp)
- On reload:
    - Given a `sessionId` (or conversationId), be able to fetch past messages and render the history.
- You can keep things simple (no auth required).

6. Robustness & Idiot-Proofing

We will try to “break” your app. Please:

- Validate input:
    - Don’t accept empty messages.
    - Handle very long messages sensibly (truncate / warn / still work).
- Make sure:
    - The backend never crashes on bad input.
    - LLM/API failures are caught and surfaced as clean error messages in the UI.
- No hard-coded secrets in the repo.
- Graceful failure > silent failure.

Non-Requirements (You don't need to do this)

- No real Shopify / Facebook / Instagram / WhatsApp integrations.
- No auth/login, unless you really want to.
- No fancy design system.
- No Kubernetes / Docker wizardry required.

If you have extra time and *want* to show off, use it to improve code quality, architecture, or UX rather than bolting on random features.

Timebox

- Designed to be doable in a **weekend (8–12 hours)**.
- Don't kill yourself over it – we care much more about **how** you build than how many extras you cram in.
- If you leave things out due to time, **document it** in the README.

Submission

Please send us:

1. **GitHub repository link** (public)
    - With all source code.
    - With clear instructions to run backend & frontend.
2. Deployed project URL
    - Some free options to deploy are Render, Vercel, Netlify and many more

Once ready, submit by filling this form.

README Must Include

- **How to run it locally**, step by step.
    - How to set up DB (migrations/seed).
    - How to configure env vars (e.g. `OPENAI_API_KEY` or `ANTHROPIC_API_KEY`).
- **Short architecture overview**:
    - How you structured the backend (layers, modules).
    - Any interesting design decisions.
- **LLM notes**:
    - Which provider you used.

    - How you're prompting it.
- **Trade-offs & "If I had more time…"** section.

How We'll Evaluate

We'll look at:

1. **Correctness**
    - Can we chat end-to-end and get sane answers from the AI?
    - Are conversations persisted?
    - Does it handle basic error cases?
2. **Code Quality & Best Practices**
    - Clean, readable, idiomatic TypeScript/JS.
    - Logical structure (separation of concerns: routes / services / data / UI).
    - Sensible naming, no obvious foot-guns.
3. **Architecture & Extensibility**
    - Is it easy to see where to plug more channels (WhatsApp, IG) or more tools later?
    - Is the LLM integration nicely encapsulated?
    - Does the schema make sense?
4. **Robustness**
    - Does it break on weird input or poor network conditions?
    - Are errors handled and surfaced nicely?
    - No obvious "one tiny change and everything explodes" moments.
5. **Product & UX Sense**
    - Is the chat experience intuitive and not annoying?
    - Are the answers phrased like a helpful support agent?
    - Does it *feel* like a small but realistic piece of a real product?

If you can build this well, you're extremely close to what you'd actually ship at Spur as a founding engineer.