VocalBridge Ops

Timebox: <5 days (please ship a working solution, ideal timeline 2-3 days)
Goal: Build a small but real "multi-tenant agent gateway" that integrates with mocked AI vendors, supports reliability features (timeouts/retries/fallback), and produces a simple usage/billing preview.

Product Context

You are building a SaaS platform where businesses (tenants) create and manage **AI agents** (voice/chat bots) that automate workflows (including billing & payments experiences). Assume there is a separate "AI team" that provides model/prompt components; your job is to integrate these into a scalable product.

This exercise is intentionally vendor-agnostic and uses **mocked AI providers**. We care about architecture, correctness, and product thinking.

What You'll Build

A backend service ("Agent Gateway") + a small React dashboard.

A) Multi-Tenant Core (hard requirement)

- A tenant can be created and issued an **API key**.
- Each tenant can create/manage multiple **agents** (bots) with config:
  - `primaryProvider`: `vendorA` | `vendorB`
  - `fallbackProvider`: optional `vendorA` | `vendorB`
  - `systemPrompt` (or equivalent)
  - optional: `enabledTools` (array)
- **Isolation**: all reads/writes must be scoped to the tenant API key. No cross-tenant access.

B) Unified Conversation API (vendor-agnostic)

Implement a minimal API surface (you choose the paths, methods, and conventions). It must support:

- Creating a **conversation session** for `(tenant, agent, customer)` and returning a `sessionId`
- Sending a **message** into an existing session and returning the assistant reply + metadata
    - Must support an `idempotencyKey` (header or body) to prevent double-charging / double-writes on retries
- Fetching a session transcript + metadata
- Fetching **usage & cost** rollups for a tenant for a date range

Persist:

- sessions
- messages/transcript
- provider call events (at least minimally)
- usage/cost events

C) AI Integration (mocked vendors; realistic behavior)

Implement a **provider adapter interface** so new vendors can be added later.

You must support two mocked vendors with intentionally different schemas + failure modes:

VendorA (chat)

- Response:
    - `outputText` (string)
    - `tokensIn` (int)
    - `tokensOut` (int)
    - `latencyMs` (int)
- Failure behavior:
    - ~10% requests return HTTP 500
    - some requests are slow (simulate latency)

VendorB (chat)

- Response:
    - `choices: [{ message: { content: string } }]`
    - `usage: { input_tokens: int, output_tokens: int }`
- Failure behavior:
    - can return HTTP 429 with `retryAfterMs`

You can implement vendors as:

- tiny local HTTP servers, or
- in-process mocks (module functions) that simulate latency/errors.

Reliability requirements

- timeouts per vendor call
- retries with backoff on transient failures (500/429/timeouts)
- **fallback**: if primary provider fails, attempt fallback provider when configured
- structured errors (don't leak stack traces)

D) Usage Metering + Billing Preview (hard requirement)

For each assistant response:

- compute cost using a pricing table (you can hardcode):
    - example pricing (feel free to choose values, but be consistent):
        - vendorA: `$0.002 / 1K tokens`
        - vendorB: `$0.003 / 1K tokens`
- store a **usage event** with:
    - tenantId, agentId, sessionId
    - provider
    - tokens in/out
    - cost
    - timestamp

Expose usage analytics:

- totals for sessions, tokens, cost
- breakdown by provider
- "top agents by cost"

E) React Dashboard (minimum viable UI)

Must include:

- "login" by API key (simple; not full auth)
- agent list + create/update basic config
- a "Try it" chat UI (text is fine; voice optional)
- a usage/analytics view (table is OK; chart optional)

Suggested Scope (2–3 days)

Choose a stack you're productive in.

- Backend: Node/TS, Python, Java, Go, etc.

- DB: SQLite/Postgres/MySQL (SQLite is fine if you document tradeoffs)
- Frontend: React (required for UI portion; minimal styling is fine)

Focus on:

- clean boundaries (tenancy, adapters)
- correctness (idempotency, cost calculation)
- clarity (docs + architecture decisions)

Deliverables

1. **Working code** in a repo you can share (GitHub link or zip).
2. `README.md`:
    - how to run backend + frontend
    - how to seed 2 tenants + 3 agents
    - sample curl commands
3. `ARCHITECTURE.md`:
    - HLD: components, tenancy isolation, scaling plan, failure handling
    - LLD: schema, adapter interface, retry/fallback logic, idempotency approach
4. Seed data:
    - at least 2 tenants
    - at least 3 agents with different configs (primary/fallback differences)
5. Tests:
    - at least a few unit tests and 1 integration test for “message -> usage billed”

Evaluation Criteria (what we look for)

- Solid application design and code quality
- Multi-tenant correctness and secure boundaries
- Extensible vendor adapter pattern
- Reliability behaviors (timeouts/retries/fallback) done thoughtfully
- Product thinking: clear UX, useful usage/billing view
- Communication: good docs, clear tradeoffs, good defaults

Bonus (pick any 2)

- **Bounty (top bonus): Voice Bot Channel Integration**

    - Add a “voice channel” so a user can speak to an agent and hear the response.
    - You may implement this using any approach you prefer. Examples (choose one):
        - A web UI that records audio in the browser and streams/uploads it to the backend
        - A phone-call integration (e.g., using a telephony provider) that forwards audio to your backend
    - Requirements:
        - Convert user audio → text (real STT API or mocked STT module)
        - Send the text through your existing session/message flow (so it’s billed, logged, and tenant-scoped)
        - Convert assistant text → audio (real TTS API or mocked TTS module)
        - Store artifacts/metadata: audio duration (if available), transcript, provider used, latency
        - UX: add a “Call/Record” or “Voice” experience in the dashboard (minimal is fine)
    - What we evaluate:
        - Clean separation of “channel” (voice vs chat) from core agent/session logic
        - Reliability and debuggability (logs, correlation IDs, failure handling)
        - Reasonable performance decisions (streaming vs upload, timeouts, retries)
- Async mode: enqueue message, return job id, completion callback/polling
- Tool/plugin framework: add one tool (e.g., `InvoiceLookup`) with audit logs
- Audio: upload audio -> mocked STT -> chat -> mocked TTS
- Observability: traces/metrics with correlation IDs
- RBAC: admin vs analyst for tenant dashboard

Submission

Send:

- repo/zip
- brief notes: what you shipped, what you’d do next with more time