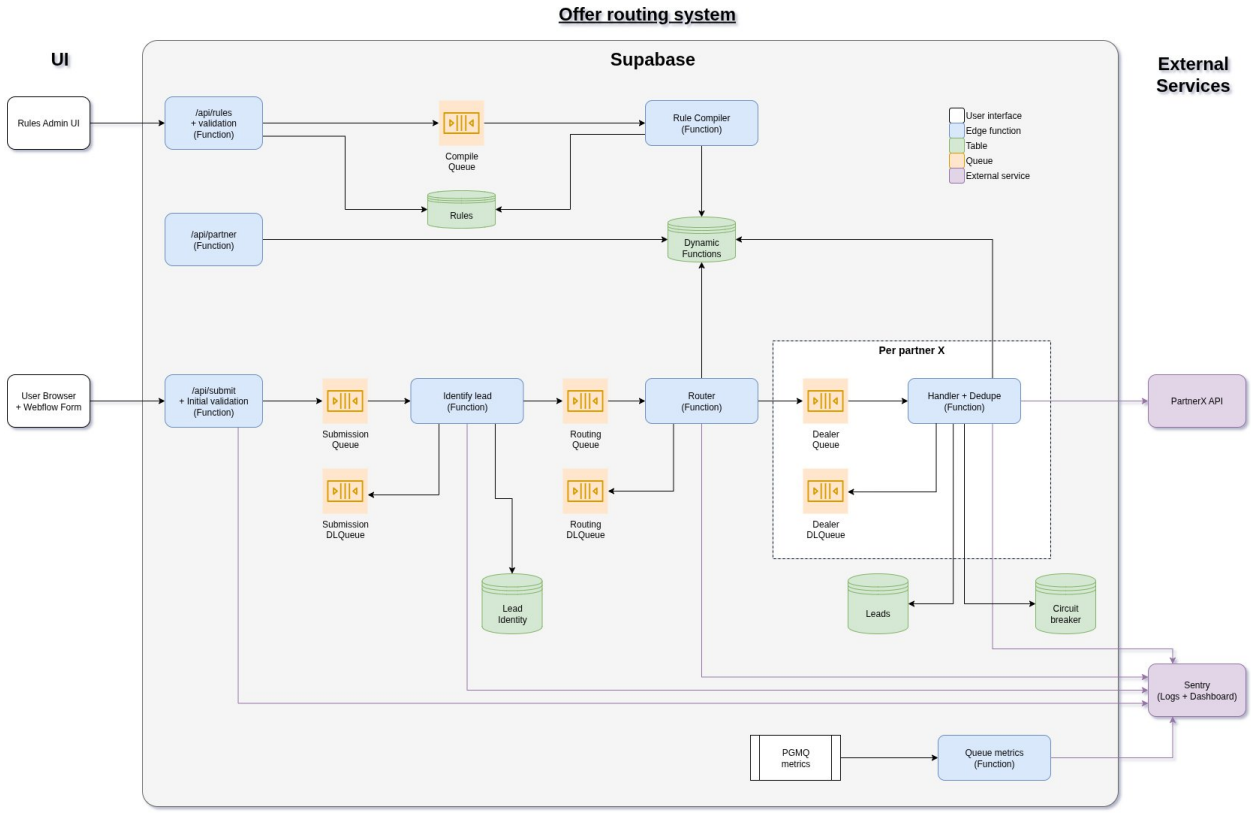
Repository + Instructions

Github (public): https://github.com/SamuelHerrera/offer-routing-system

Project uses supabase CLI

- Start with the login to supabase (deno task login)
- Link with a empty project (deno task link)
- Then run the migration (deno task db:push)
- Then call the deploy command (deno task deploy)
- Then register rules with the rule endpoint
- Then register dealer dedupe and handler functions
- Lastly call submit endpoint to trigger the pipeline execution



Part 1: Core Form Workflow (Foundational)

Tasks:

1. Describe your architecture and approach (front-end → back-end → DB → external API).
    a. Architecture diagram is available at design.drawio but in general sense the idea is to split in to stages separated by queues to handle the back-pressure and potential slow actors.
    b. Store the rules as db records and then compile the predicates at to build a decision tree that will be evaluated/instantiated for best routing performance

    c. Create individual dealer queues and handlers to avoid propagate slowness to the system from lazy clients
    d. Implement a cron supervisor to keep workers running an pulling from queues
    e. Use dead letter queues to quarantine failed messages
    f. Use retry approach (deno) for task steps before offloading to DLQueue
    g. Use a circuit breaker in client implementations to react to service outages
2. Show sample code or pseudo code for validation, database insertion, and API call.
    a. To validate duplicates in lead identities using basic info I implemented a logic to match by email or phone (assuming this is sufficient) and save aliases if a matching <partial> record is found, so an alias is a separate record with a different email/phone value that references (fk) a main entry. **Check identify-worker function**
    b. To validate duplicated submissions (for an specific lead identity, allowing a same lead to push events to different dealers), I decided for a composite string key approach (very like Dynamo) that is built by a custom <per dealer> function. And we can dynamically register a dealer by creating the queues and defining a dedupeKeyFn and a handlerFn along with the rules to redirect. **Check router-worker, rule-compiler and _shared/rule.ts files**
    c. For database insertions **_shared/worker.ts** handles worker states in a table, also **identify-worker** file is managing lead identities
3. Explain how you would:
    a. Handle errors (invalid data, failed API call).
        i. If its at controller level, return an appropriate status code 400
        ii. If it happens at worker level we move the message to DLQ
        iii. Check API responses to act accordingly, like handling too many request code with a retry vs opening the circuit breaker on multiple incidents
    b. Prevent duplicate submissions.
        i. There are two levels here, we first de-duplicated the lead identity and secondly we act on the actual form information by generating a composite key to uniquely identify it
        ii. The order of the instructions is also particular as duplicate messages can exist in the queues but once a handler acts on a record we first persist it with a pending state so even other instances will not process the duplicated messages.
    c. Log or track submission results.
        i. Submission results are written back to the lead record
4. Describe how you would use AI to accelerate this task (e.g., generate boilerplate, validation schema, test data) while maintaining accuracy and security.
    a. Asking to generate integration tests to easily test individual portions of the services
    b. Create the db schema and migrations

Part 2: Offer Routing & Logic Layer (Intermediate)

Tasks:

1. Extend your system to decide which partner API to send each lead to, using simple routing logic such as:
    a. If interest = "Education", send to EDU API.
    b. If country = "US" and interest = "Finance", send to FIN API.
    c. Otherwise, send to fallback API.
    - Initially we can store this as individual records or a JSON document that describes the decision tree
    - Ultimately I planned for a UI to handle the predicates and routing so it gets compiled to js code when updated and evaluated at runtime.
2. Provide pseudo code, schema, or a short code sample that shows how you would implement this routing.
    a. Get all rules from the DB, then iterate all the rules and its conditions to build a tree, do some optimizations based on frequency and generate a JS string of a function that accepts a message with the form data and returns the route string to take
3. Show how you would test this logic locally and handle routing failures.
    a. Do an integration test with multiple scenarios since the generate function is pure JS
    b. Routing failures are just moving messages to DQL where we can reprocess them when rule or route is fixed.
    c. Check _shared/rules.ts
4. Include examples of AI prompts or assistance you'd use to speed up development (for example, generating test cases or mock APIs).

Part 3: Dynamic Rules & Monitoring

Tasks:

1. Describe how you'd make routing rules data-driven (e.g., stored in a database table or JSON configuration).
    a. Create a rules table, each record contains a json for the predicates that the user created (including grouping operators)
    b. Then process them in to a single routing function string
    c. Load/evaluate the function string at runtime
2. Design a simple logging or monitoring approach using Supabase logs, a dashboard, or an external tool.
    a. Send everything to Sentry, incluide context data
    b. Build Sentry dashboards using the metadata
    c. Track failures, execution times and counters
3. Provide an example schema or JSON structure for these dynamic rules.

```
[
  {
    "id": "11111111-1111-1111-1111-111111111111",
    "name": "US high-score to Dealer A",
    "priority": 10,
    "predicate_json": {
      "and": [
        { "field": "payload.country", "op": "===", "value": "US" },
        { "field": "payload.score", "op": ">=", "value": 750 }
      ]
    },
    "route_name": "dealer_a",
    "enabled": true,
    "created_at": "2025-10-15T00:00:00.000Z",
    "updated_at": "2025-10-15T00:00:00.000Z"
  }
]
```

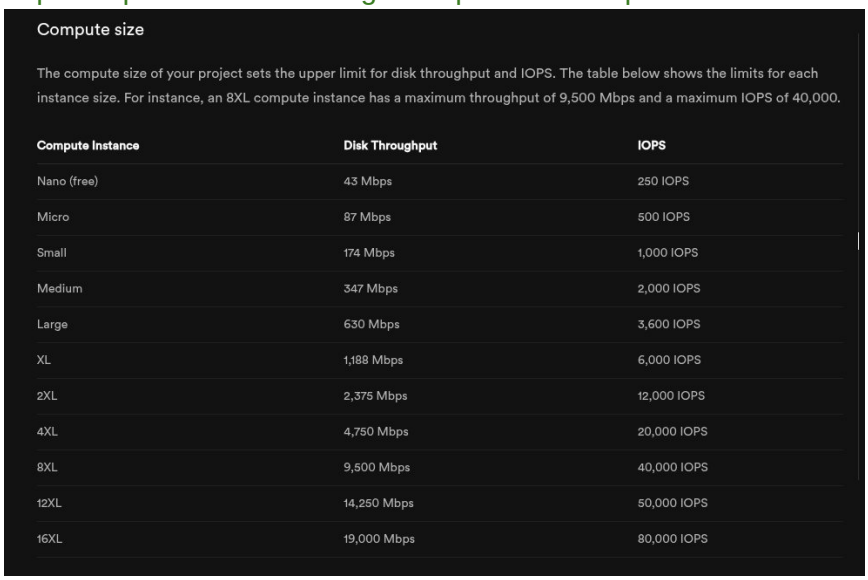
4. Write a short message (3–4 sentences) you'd send to your internal team (PM + Data) explaining how to update rules and view metrics.
    a. Go to the rule web app, login, it should list all the rules as a table and as a tree (for better visualization), add/remove/edit the rule name, the predicates (property + operator + expected) and the route. Once saved it will automatically compile the rules and start using it

Part 4: Scaling & System Design

Tasks:

1. Outline a high-level system design that could support this scale. Include queues, retries, caching, and monitoring if relevant.
    a. This aims to handle around 4 (3.47) requests per second (assuming an 8hours window, so 100000 req / 8hours / 60mins / 60secs = 3.47 req/sec
    b. Each request uses at least 6 to 10 IOPS counting queues and db reads/writes
    c. So to be on the safe side we consider 4 requests/sec * 10 IOPS/req = 40 IOPS/sec and nano instances can handle 250 IOPS

d. https://supabase.com/docs/guides/platform/compute-and-disk



Compute size

The compute size of your project sets the upper limit for disk throughput and IOPS. The table below shows the limits for each instance size. For instance, an 8XL compute instance has a maximum throughput of 9,500 Mbps and a maximum IOPS of 40,000.

Compute Instance	Disk Throughput	IOPS
Nano (free)	43 Mbps	250 IOPS
Micro	87 Mbps	500 IOPS
Small	174 Mbps	1,000 IOPS
Medium	347 Mbps	2,000 IOPS
Large	630 Mbps	3,600 IOPS
XL	1,188 Mbps	6,000 IOPS
2XL	2,375 Mbps	12,000 IOPS
4XL	4,750 Mbps	20,000 IOPS
8XL	9,500 Mbps	40,000 IOPS
12XL	14,250 Mbps	50,000 IOPS
16XL	19,000 Mbps	80,000 IOPS

e. Also we must consider message size and Wall time or CPU limits from supabase to find the sweet spot for batch size and retry policies
f. With the queues we handle backpressure
g. Retries are in place with deno lib and Circuit Breaker for API calls
h. DLQueues are last defense line to hold messages so they can be re published on demand

2. Explain how you would ensure data consistency between your database, external APIs, and analytics tools.
    a. First all data gets persisted first internally then we call external services so no changes are allowed and data is idempotent
    b. Analytics are always behind depending on the external service we use but they are just gathering info that its already persisted
3. Suggest AI-powered automation you'd add — for example, daily performance summaries, anomaly detection, or automatic routing optimization.
    a. Queue metrics function is for this purpose, it queries the queue sizes and health and can act depending on conditions, like spawning more workers (or decreasing them) of certain type, we can implement offloading strategies when outages happens (like redirect to db for later execution on a cold start)
    b. Summaries per day, hour or outage including AI to humanize the information