

Real-Time CRM Lead Processing & Notification System – Detailed Guide

1 Problem Statement & Goals

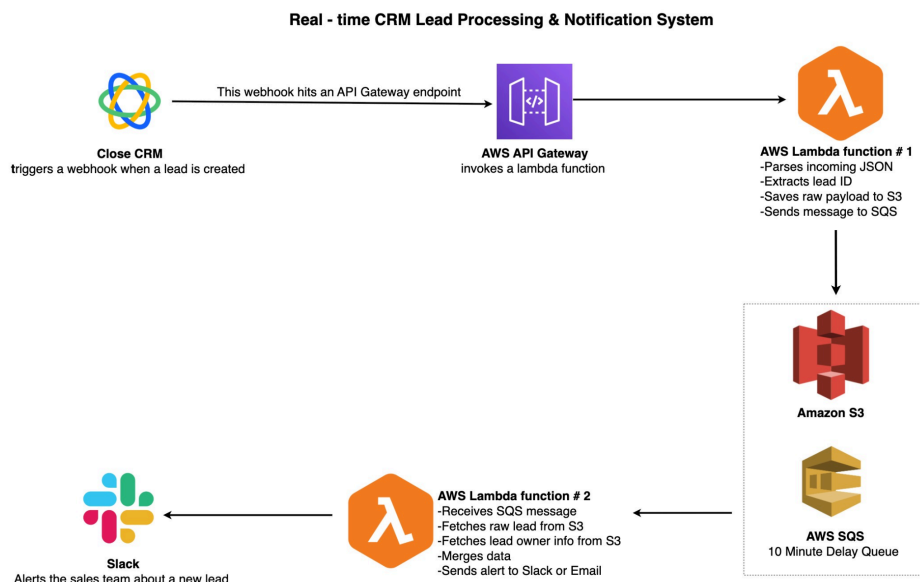
In many sales organizations the act of creating a lead in the CRM is only the first step; the lead still needs to be assigned to the correct owner. Without automation there is often a delay between the lead creation and the assignment. If the sales team act before the owner field is populated, leads may be contacted by the wrong person, leading to confusion or missed revenue. The goal of this project is to:

- Capture lead creation events from **Close CRM** in real-time via webhooks.
- Buffer each event for roughly 10 minutes, allowing the CRM to assign the lead owner.
- Enrich the original event with the assigned owner's details by performing a lookup in a separate S3 bucket..
- Persist both the raw and enriched events in Amazon S3 for later analysis.
- Notify the sales team immediately after enrichment via Slack or email, including all required lead details in a consistent format.

The system must be fully serverless, handle multiple leads in parallel without race conditions and provide logs for debugging and auditing.

2 Solution Overview

The architecture leverages managed AWS services to provide a robust, scalable and cost-effective pipeline. The flow is summarised in the diagram below



Components at a Glance

#	Component	Purpose
1	Close CRM	Emits a webhook every time a lead is created.
2	API Gateway (HTTP API)	Exposes a public HTTPS endpoint that receives the webhook and triggers the first Lambda.
3	Lambda #1 – webhook-ingest	Parses the webhook payload, stores it in S3 and sends a message to the SQS delay queue.
4	S3 (Raw Bucket)	Stores raw lead events and, later, enriched events.
5	SQS Delay Queue	Buffers the message for 10 minutes, ensuring the lead owner has been assigned.
6	Lambda #2 – enrich-alert	Retrieves the raw event from S3, fetches owner data from a public S3 bucket, merges them, writes the enriched object and sends a notification.
7	Slack	Serves as the primary notification channel for the sales team.

3 Pipeline Walkthrough

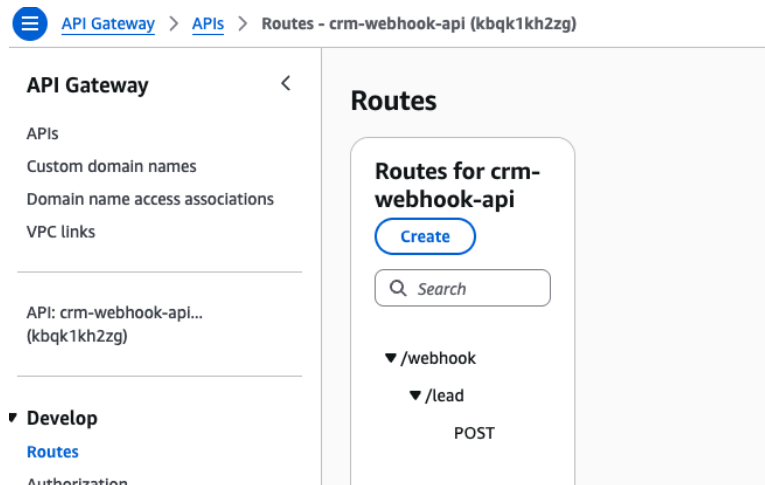
3.1 Lead Creation & Webhook Trigger

When a new lead is created in Close CRM the system immediately triggers a webhook. The payload includes details such as the lead's display name, status label, creation timestamp and the crucial `lead_id`. The `lead_id` may appear either at `event.lead_id` or nested under `event.data.id`, and the first Lambda accounts for both possibilities.

3.2 API Gateway — Receiving the Webhook

The webhook is delivered over HTTPS to Amazon API Gateway. A simple HTTP API is configured with a POST route such as `/webhook/lead`. The route has no authorization because Close uses a secret signing key. The route forwards the request to the

Lambda. The API Gateway configured

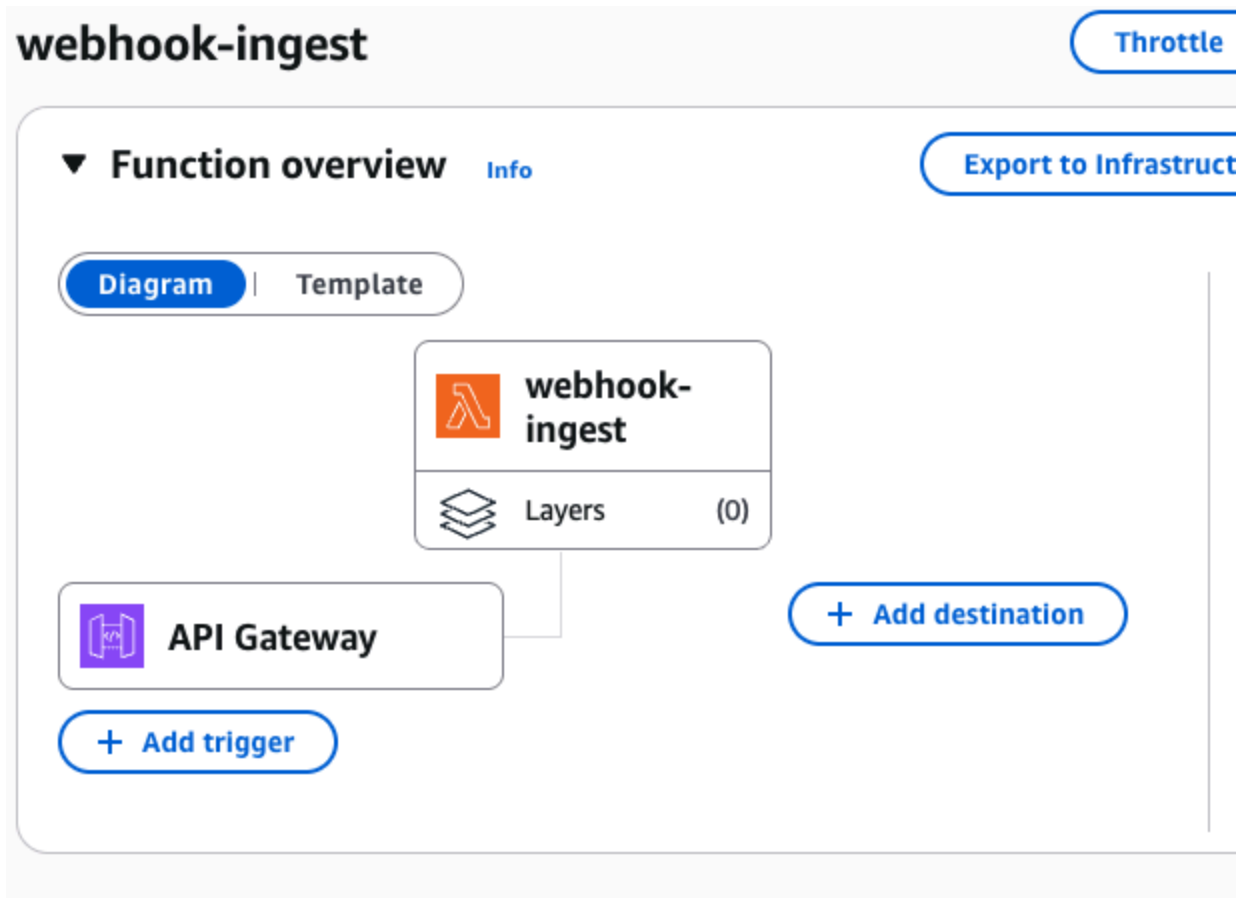


webhook-ingest routes page in shows the path:

3.3 Lambda #1 (webhook-ingest) — Ingest & Queue

The first Lambda function is invoked with the JSON payload. Its logic is straightforward:

1. **Parse the payload.** The event body is parsed as JSON. The function retrieves the `lead_id` from either `event.lead_id` or `event.data.id`. If neither exist, the function can raise an error (the provided code also allows custom fallback logic).
2. **Persist the raw event to S3.** The entire JSON payload is serialized with indentation and stored in the raw events bucket with the key `crm_event_{lead_id}.json`. The naming convention is mandated by the project requirements.
3. **Send a message to SQS.** The function posts a message containing the `lead_id` and S3 key to the delay queue. Because the queue's default delay is set to 10 minutes, no per-message delay is specified. The response from SQS is logged.



3.4 Amazon SQS — Delay Queue

SQS acts as a buffer between the two Lambdas. By creating a **delay queue** (a standard queue with a non-zero Delay Seconds), messages are hidden from consumers until the delay period expires. This ensures that the second Lambda does not fire until enough time has passed for the CRM system to assign a lead owner. The queue list shows a queue named `lead-processing-delay-queue` created on 2025-06-05 with no messages available.

Drilling into the queue reveals its URL, ARN and access policy. The policy grants only the Lambda's IAM role permission to send and receive messages:

Amazon SQS > Queues > lead-processing-delay-queue

ⓘ

🔗

📘 SQS fair queues is now available for standard queues.
Automatically manage noisy neighbors in your queues with fair queues, a new feature that limits noisy neighbor impact across all message groups. Add a group identifier to your messages, and SQS re-orders messages to ensure no single tenant impact the time in queue for any other tenants.

Learn more 🔗

✕

lead-processing-delay-queue

Edit

Delete

Purge

Send and receive messages

Start DLQ redrive

Details Info

Name

🔗 lead-processing-delay-queue

Type

Standard

ARN

🔗 arn:aws:sqs:us-east-1:654654165755:lead-processing-delay-queue

Encryption

Amazon SQS key (SSE-SQS)

URL

🔗 https://sqs.us-east-1.amazonaws.com/654654165755/lead-processing-delay-queue

Dead-letter queue

-

▶ More

<

Queue policies

Monitoring

SNS subscriptions

Lambda triggers

EventBridge Pipes

Dead-letter queue

Tags

>

Access policy Info

Edit

Define who can access your queue.

```
{
  "Version": "2012-10-17",
  "Id": "__default_policy_ID",
  "Statement": [
    {
      "Sid": "__owner_statement",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::654654165755:root"
      },
      "Action": "SQS:*",
      "Resource": "arn:aws:sqs:us-east-1:654654165755:lead-processing-delay-queue"
    }
  ]
}
```

3.5 Lambda #2 (enrich-alert) — Enrichment & Notification

After the 10-minute delay the message becomes visible to the enrich-alert Lambda. The function processes messages in order, but SQS allows multiple invocations in parallel if necessary.

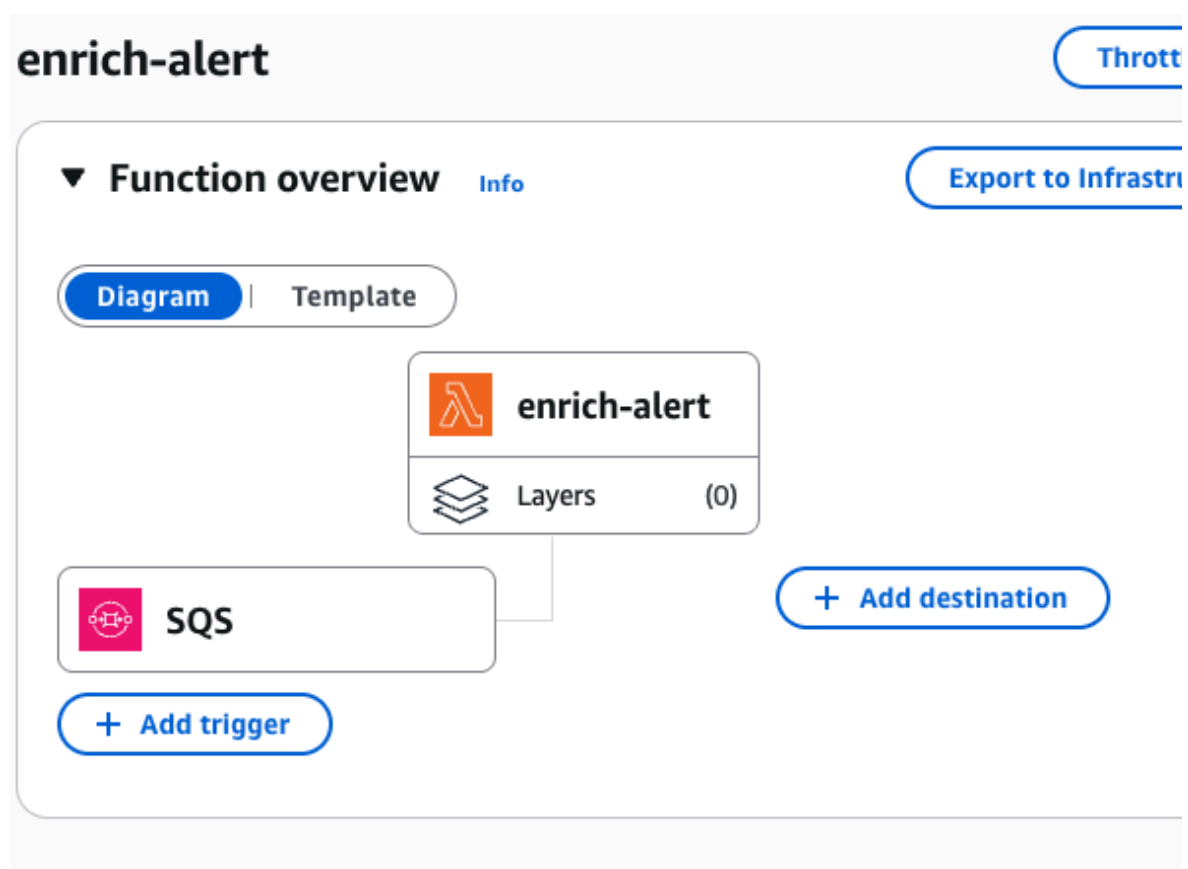
The key steps performed by this function are:

1. **Read the SQS message.** Each record contains the `lead_id` and the S3 key of the raw event.
2. **Download the raw event.** The function performs a `GetObject` request against the raw bucket to retrieve the JSON.
3. **Fetch the lead owner data.** A second public S3 bucket contains JSON files keyed by `lead_id` and holding owner details such as the lead's email, owner and funnel.

The function uses `urllib.request` to fetch the public URL and includes retry logic with exponential backoff to cope with transient 403 or network errors.

4. **Merge the data.** The raw event and the owner record are merged into a single document. The merged object is then saved back to S3 under the `enriched/` prefix, preserving the original file name for traceability.
5. **Send an alert.** A Slack message is assembled with the required fields (name, lead ID, created date, status label, email, lead owner and funnel) and posted to the incoming webhook URL. The Slack API uses an HTTP POST with a JSON body and a Content-Type header. If you prefer email, replace this section with a call to Amazon SES or another email service.

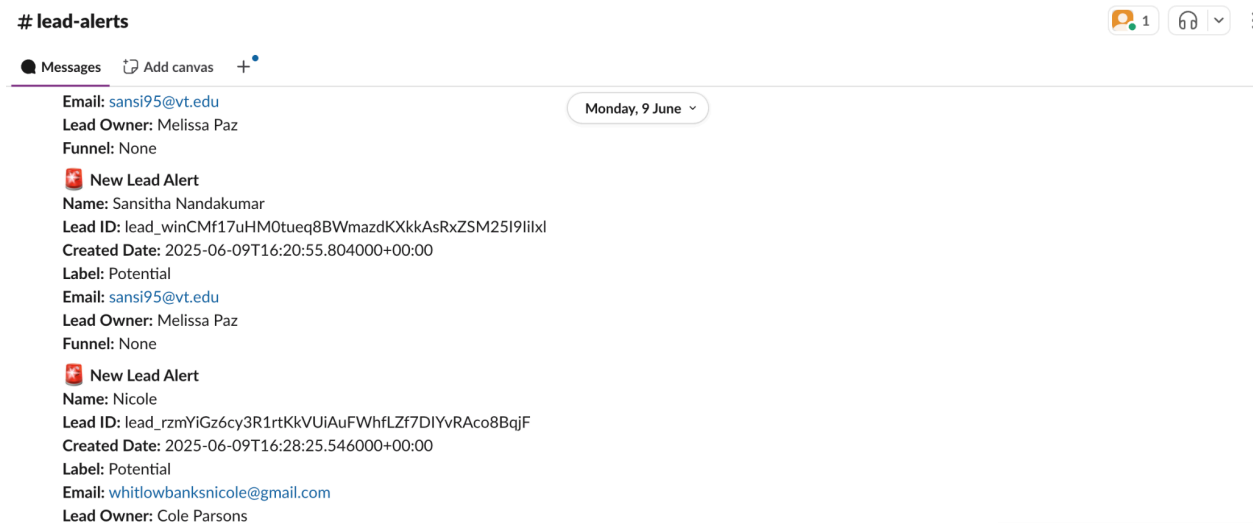
The function overview in AWS shows SQS as the trigger for this Lambda:



3.6 Slack Notifications

Slack provides an easy way to notify your sales team when a new lead is ready. After you create an incoming webhook in your Slack workspace you obtain a unique URL. This URL is set as the `SLACK_WEBHOOK` environment variable in the second Lambda. When the Lambda posts a message to this webhook Slack delivers it to the configured channel.

The messages include all of the fields specified in the project brief and are formatted using simple newline separators. The screenshot below shows several sample notifications in a #lead-alerts channel:



Each alert contains the lead's **Name**, **Lead ID**, **Created Date**, **Label**, **Email**, **Lead Owner** and **Funnel**.

3.7 Enriched Data for Auditing

In addition to real-time notifications the project persists the enriched events back into S3. Storing the combined data under `enriched/` allows downstream analytics to compare raw versus enriched records, audit owner assignment delays and feed dashboards without re-processing the data. Because the file names mirror the raw objects, they can be joined programmatically on `lead_id`.

4 Implementation Details

4.1 IAM Roles & Permissions

Two IAM roles are created—one for each Lambda function. Both roles include the **AWSLambdaBasicExecutionRole** policy for logging. In addition:

- The `webhook-ingest` role requires `s3:PutObject` permission on the raw bucket and `sqs:SendMessage` permission on the delay queue.
- The `enrich-alert` role requires `s3:GetObject` and `s3:PutObject` on the raw bucket. Because the queue triggers the function, SQS permissions are not needed.

Always scope your IAM policies down to the exact ARNs of the resources instead of using wildcards.

4.2 Environment Variables



Use environment variables to decouple configuration from code:

- `SQS_QUEUE_URL` – The full URL of your delay queue (Lambda #1).
- `RAW_BUCKET` – The name of the S3 bucket storing raw and enriched events (Lambda #2).
- `SLACK_WEBHOOK` – The Slack incoming webhook URL (Lambda #2).

You can extend this pattern to support multiple notification channels or to parameterise the lookup bucket.

4.3 Error Handling & Retries

Robustness is crucial when dealing with external webhooks and network calls. The following patterns are used in the provided code:

- **Exponential backoff for HTTP requests.** The `fetch_with_retries` helper tries up to 10 times to fetch the owner file, doubling the delay after each retry. It explicitly handles 403 errors—which may occur due to S3 eventual consistency—by waiting and retrying.
- **Graceful SQS sending.** The first Lambda wraps the call to `send_message` in a try/except block and logs a failure without stopping the function. If SQS is unavailable the event is still stored in S3.
- **Structured logging.** Both functions print human-readable messages prefaced with  or  to indicate success or failure. These show up in CloudWatch Logs and simplify debugging.

4.4 Scaling & Concurrency

Because the pipeline is entirely serverless it can scale to handle hundreds or thousands of leads without modification. Amazon SQS distributes messages across multiple Lambda invocations and ensures that each message is processed exactly once. The 10-minute delay is per-message, meaning that simultaneous leads do not interfere with each other. If your sales volume spikes you can increase the Lambda concurrency limit to improve throughput.

5 Evaluation & Success Criteria

During development and testing you can verify that the system meets the project's evaluation criteria:

1. **README & architecture diagram** – The provided README contains a step-by-step setup guide and the architecture diagram appears at the top of both documents.
2. **Modular code** – The two Lambda functions are concise and separated by responsibility: the first ingests and queues, the second enriches and alerts.

3. **Alert delivery** – Slack notifications show the fields required by the specification: name, lead ID, created date, label, email, lead owner and funnel.
4. **Error handling and logging** – Both Lambdas catch and log exceptions; the second function retries network failures with exponential backoff.
5. **Success criteria** – Leads are ingested and saved in S3 within seconds, processed only after a 10-minute delay, enriched correctly by `lead_id`, and delivered as real-time notifications. The use of SQS prevents race conditions and allows multiple leads to be processed concurrently.

6 Conclusion

This project demonstrates how to build a practical, event-driven integration between a SaaS CRM and a communication platform using AWS serverless services. By isolating responsibilities into small functions and leveraging managed services like SQS and API Gateway, the system achieves reliability and scalability with minimal operational overhead. The modular design also makes it easy to adapt—should you wish to enrich leads with data from another source, send notifications by email, or integrate a different CRM, only small changes are needed.
