

Auto-Reply Email System for Customer Complaints

This document provides a complete technical overview of the automated email reply system. It details the motivation, architecture, and implementation steps required to build a real-time system using Google Pub/Sub and a Large Language Model (LLM).

1. Motivation

1.1. Problem Statement

- Our current method of handling customer support emails is entirely manual. Each email is read and answered by the agent around 5000 emails per day, regardless of its complexity or repetitiveness.
- This process is inefficient and costly. It leads to slow response times for customers, inconsistent answers due to a lack of standardization, and high operational costs as we need more staff to handle a growing volume of emails.

1.2. Objective

- The goal is to build an intelligent automated system that can instantly categorize and reply to customer emails.
- This will dramatically improve response times, ensure every customer receives a consistent and accurate reply, and free up our support agents to focus on complex issues that require a human touch. This allows us to scale our agents capabilities without proportionally increasing costs.

2. High-Level Flow Process

2.1. Step-by-Step Flow

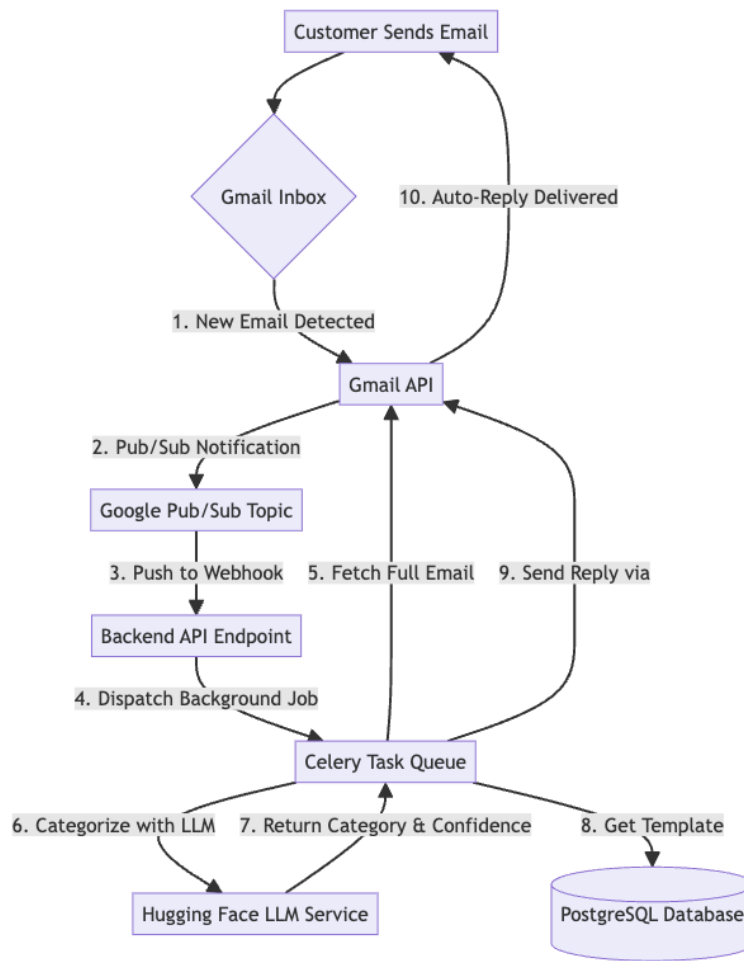
1. A customer **sends an email** to our support email.
2. The Gmail API detects the new email by **Gmail Watch** and instantly sends a notification to our **Google Pub/Sub topic**.
3. **Pub/Sub pushes** this notification to our backend **webhook** endpoint.
4. **Our backend receives** the notification and **triggers a background task** to fetch the full email content (subject, sender, body) from the Gmail API using the message ID.
5. This email content is then passed to a **Large Language Model (LLM)** to determine its category (e.g., "Registration Problem," "Payment Issue").
6. The system checks if the **LLM's confidence** in the prediction is above our defined threshold (e.g., 85%).
7. If confidence is high, the system retrieves the appropriate **pre-defined response template** from our database based on the predicted category.
8. Finally, the system uses the **Gmail API to send** this templated response back to the

customer as a threaded reply to the original email.

2.2. Diagram

graph TD

```
A[Customer Sends Email] --> B{Gmail Inbox};
B -->|1. New Email Detected| C[Gmail API];
C -->|2. Pub/Sub Notification| D[Google Pub/Sub Topic];
D -->|3. Push to Webhook| E[Backend API Endpoint];
E -->|4. Dispatch Background Job| F[Celery Task Queue];
F -->|5. Fetch Full Email| C;
F -->|6. Categorize with LLM| G[Hugging Face LLM Service];
G -->|7. Return Category & Confidence| F;
F -->|8. Get Template| H[(PostgreSQL Database)];
F -->|9. Send Reply via| C;
C -->|10. Auto-Reply Delivered| A;
```



3. Product Requirements

- **Self-Service Operation:** The system must run automatically without needing manual intervention for each email.
- **Template Management:**
The template management interface will be integrated into the new crm (crmui), communicating with the backend via REST APIs. Authentication will leverage the existing login flow through MVP with redirection, so no additional development is required on the authentication side.

The template management interface in **crmui** will consist of two main components. While final UI design will likely depend on input from the design team, the essential structure is as follows:

- **Template Table View**

A searchable and filterable table that displays all existing templates. Users can perform CRUD operations directly from the list, including editing, disabling, or deleting templates.

- **Template Form**

A form used for creating new templates or editing existing ones. It will utilize the existing data fields and include components such as a rich text editor (WYSIWYG) for editing the email body.

- **Confidence Threshold:** The system must only send a reply if the LLM is sufficiently confident in its categorization.

4. System Requirements

4.1. Software

- **Backend Framework: Django (Python)**

- **What it is:** A high-level Python web framework.
- **Why we use it:** It encourages rapid development, has a robust ecosystem, and includes features like an admin panel out-of-the-box, which is perfect for managing our reply templates.

- **NLP Model: Hugging Face Transformers**

- **What it is:** A library providing state-of-the-art machine learning models for Natural Language Processing (NLP).
- **Why we use it:** It gives us access to powerful, pre-trained language models that can categorize text without needing to train our own model from scratch.
-

4.2. Infrastructure

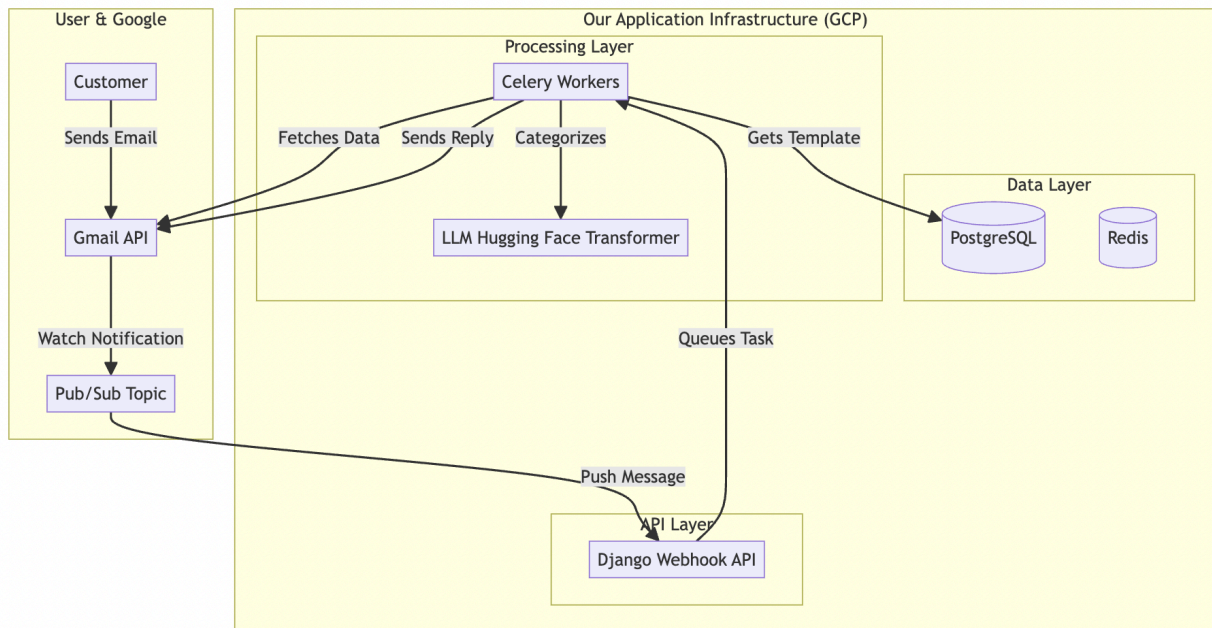
- **Server: Google Cloud Platform (GCP)**
 - **What it is:** A suite of cloud computing services.
 - **Why we use it:** It provides all the necessary services (Pub/Sub, IAM, Watch) and integrates seamlessly with the Gmail API.
- **Database: PostgreSQL**
 - **What it is:** An open-source object-relational database system.
 - **Why we use it:** It's reliable, scalable, and well-supported by Django. It will store our reply templates and user authentication tokens.
- **Caching: Redis**
 - **What it is:** An in-memory data store.
 - **Why we use it:** We can use it to cache frequently accessed templates or user credentials to reduce database load and improve performance.
- **Task Queue: Celery**
 - **What it is:** A distributed task queue system.
 - **Why we use it:** Processing an email (fetching, categorizing, replying) can take a few seconds. Celery allows us to run this process in the background, so our API can respond instantly to the Pub/Sub notification without making Google wait.

4.3. Monitoring

- **API & Performance: Datadog**
 - **What it is:** A monitoring and analytics platform.
 - **Why we use it:** To track the performance of our API endpoints, monitor the processing time of our Celery tasks, and set up alerts for any failures.
- **Error Tracking: Sentry**
 - **What it is:** An open-source error tracking tool.
 - **Why we use it:** It provides real-time error tracking and reporting for our Django application, allowing us to fix bugs before they impact many users.

5. System Architecture

5.1. High-Level Architecture



graph TD

```

subgraph "User & Google"
    User[Customer]
    Gmail[Gmail API]
    PubSub[Pub/Sub Topic]
end
  
```

```

subgraph "Our Application Infrastructure (GCP)"
    direction LR
    subgraph "API Layer"
        Webhook[Django Webhook API]
    end
    end
    subgraph "Processing Layer"
        Celery[Celery Workers]
        LLM[LLM Service]
    end
    end
    subgraph "Data Layer"
        DB[(PostgreSQL)]
        Cache[(Redis)]
    end
    end
end
  
```

```

User -- "Sends Email" --> Gmail;
  
```

```

Gmail --"Watch Notification"--> PubSub;
PubSub --"Push Message"--> Webhook;
Webhook --"Queues Task"--> Celery;
Celery --"Fetches Data"--> Gmail;
Celery --"Categorizes"--> LLM;
Celery --"Gets Template"--> DB;
Celery --"Sends Reply"--> Gmail;

```

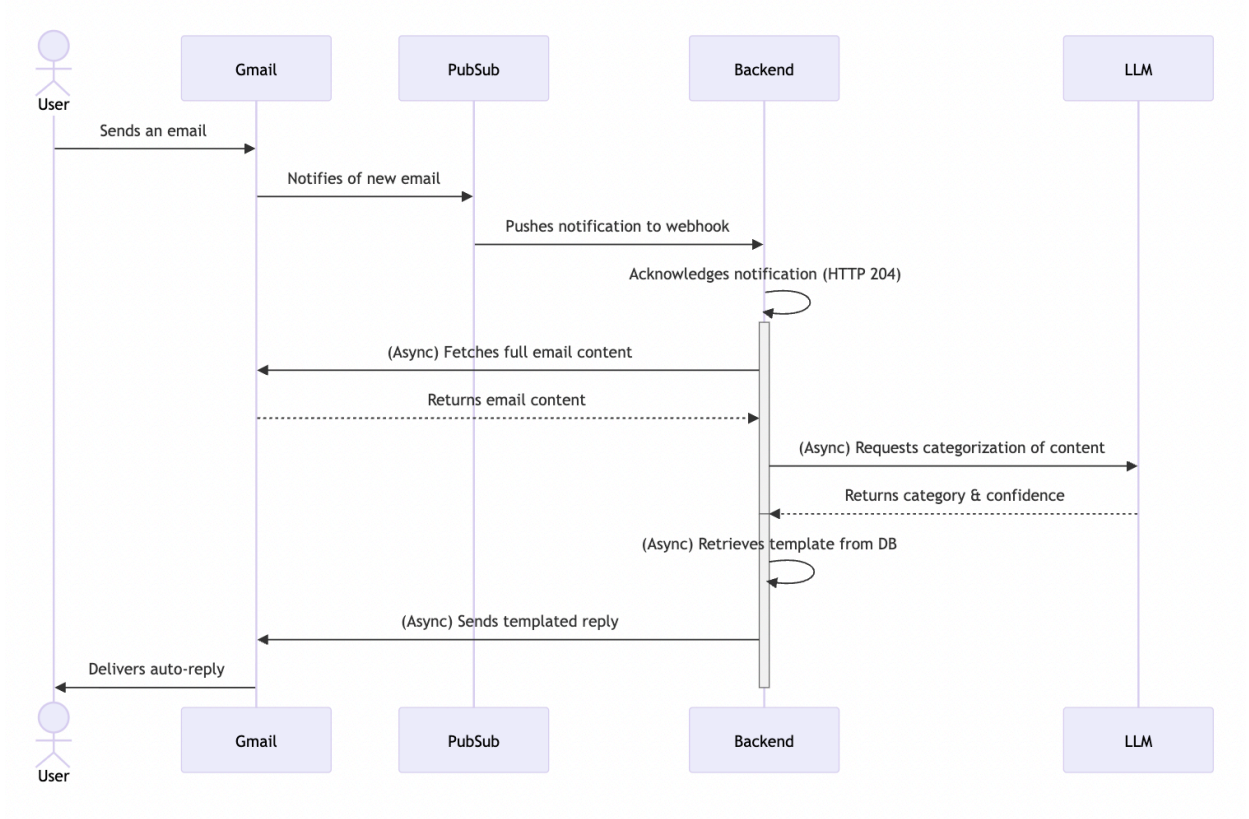
5.2. Data Modeling

Table Specifications

Table Name	Column	Data Type	Description
autoreply_template	id	BigAutoField	Primary Key
	category	CharField(100)	The unique category name (e.g., registration_problem).
	subject	CharField(255)	The subject line for the reply email.
	content	TextField	The body of the email reply.
	is_active	BooleanField	A flag to enable or disable this template.
gmail_token	id	BigAutoField	Primary Key
	user_email	EmailField	The email address of the monitored account.
	credentials	JSONField	The encrypted OAuth2 token from Google.
complaint_email_log	id	BigAutoField	Primary Key
	body	TextField	The content of email
	email_from	CharField(255)	Customer's email

	confidence_score	DecimalField	Weight or Score resulted by transformer
	predicted_category	CharField(100)	Category resulted by LLM
	actual_category	CharField(100)	Category that filled by agent or human
	template_response_id	IntegerField	ID of predefine template response

5.3. Sequence Diagram



```
sequenceDiagram
    actor User
    participant Gmail
    participant PubSub
    participant Backend
```

participant LLM

User->>Gmail: Sends an email

Gmail->>PubSub: Notifies of new email

PubSub->>Backend: Pushes notification to webhook

Backend->>Backend: Acknowledges notification (HTTP 204)

activate Backend

Backend->>Gmail: (Async) Fetches full email content

Gmail-->>Backend: Returns email content

Backend->>LLM: (Async) Requests categorization of content

LLM-->>Backend: Returns category & confidence

deactivate Backend

activate Backend

Backend->>Backend: (Async) Retrieves template from DB

Backend->>Gmail: (Async) Sends templated reply

Gmail->>User: Delivers auto-reply

deactivate Backend

6. API Contract

1. OAuth2 Start

- **Description:** Redirects the user to Google's consent screen to grant permission.
- **Endpoint:** GET /api/oauth2/start
- **Example Request:** A simple browser request to the endpoint.
- **Example Response:** An HTTP 302 Found redirect.

HTTP/1.1 302 Found

Location:

https://accounts.google.com/o/oauth2/v2/auth?response_type=code&client_id=...&scope=...&state=...&redirect_uri=...

2. OAuth2 Callback

- **Description:** The endpoint Google redirects to after user consent. It exchanges the code for a token.
- **Endpoint:** GET /api/oauth2/callback
- **Example Request (from Google):**
GET /api/oauth2/callback?code=4/0Ad...&scope=...&state=...

- **Example Response (from our API):**

{


```
    "status": "success",
    "message": "Authentication successful. You can now close this
window."
}
```

3. Register Watch

- **Description:** Tells Gmail to start sending notifications for the authenticated user's inbox.
- **Endpoint:** POST /api/gmail/watch/register
- **Example Request Body:** (Empty)
- **Example Response (Success):**

```
{
  "historyId": "1234567",
  "expiration": "1672531199000"
}
```

4. Stop Watch

- **Description:** Tells Gmail to stop sending notifications.
- **Endpoint:** POST /api/gmail/watch/stop
- **Example Request Body:** (Empty)
- **Example Response (Success):** An empty response with HTTP status 204 No Content.

5. Pub/Sub Callback

- **Description:** The main webhook that receives notifications from Google Pub/Sub.
- **Endpoint:** POST /api/webhooks/pubsub
- **Example Request Body (from Google):**

```
{
  "message": {
    "data":
"eyJlbWFpbEFkZHZHJlc3MiOiJzdXBwb3J0QGV4YW1wbGUuY29tIiwiaGlzdG9yeUlkIj
oiMTIzNDU2OCJ9",
    "messageId": "987654321",
    "publishTime": "2025-07-21T14:12:00Z"
  },
  "subscription":
"projects/project-id/subscriptions/subscription-name"
}
```

Note: data is a base64-encoded JSON string:

```
{"emailAddress":"support@example.com","historyId":"1234568"}
```

- **Example Response (Success):** An empty response with HTTP status 204 No Content to acknowledge receipt.

7. Implementation Plan: Task Breakdown

#	Task	Stack	Description	ETA	PIC
1	GCP Setup	DevOps	Configure Pub/Sub topic, subscription with DLQ, Service Accounts, and IAM roles.		Alvian Supriadi
2	Data Models	Backend	Create Django models for AutoReplyTemplate and GmailToken.		Muhammad Rizqi ...
3	OAuth2 Flow	Backend	Implement /oauth2/start and /oauth2/callback endpoints for Google authentication.		Alvian Supriadi
4	Gmail Watch API	Backend	Implement /gmail/watch/register and /gmail/watch/stop endpoints.		Alvian Supriadi
5	Watch Renewal Task	Backend	Create a scheduled Celery task to automatically renew the watch every 6 days.		Alvian Supriadi
6	Pub/Sub Webhook	Backend	Develop the /webhooks/pubsub endpoint to receive and parse notifications.		Alvian Supriadi

7	Email Processing Task	Backend	Create the main Celery task to fetch email, call the LLM, check the threshold, get the template, and send the reply.		
8	Template Management API	Backend	Develop REST API endpoints to support CRUD operations for auto-reply templates.		Muhammad Rizqi ...
9	Template Management UI	Frontend Web	Develop the user interface for managing auto-reply templates within the new crm.		Lewis Spencer
10	Unit & Integration Tests	Backend	Write tests for all new services, API endpoints, and the email processing logic.		Muhammad Rizqi ...

8. Research Result

8.1. Gmail & Pub/Sub Integration

a. Service Account & Credentials

- **What it is:** A special type of Google account for an application, not a person. It has its own identity and a JSON key file for authentication.
- **Why we use it:** It allows our backend application to securely and programmatically authenticate with Google Cloud services without needing to manage or expose a human user's credentials.
- **Console:** IAM & Admin > Service Accounts > Create Service Account. Follow the prompts and download the JSON key.
- **gcloud Shell:**

```
# Create the service account
gcloud iam service-accounts create auto-reply-service
--display-name="Auto Reply Service"

# Create and download the key file
```

```
gcloud iam service-accounts keys create credentials.json \
--iam-account=auto-reply-service@PROJECT_ID.iam.gserviceaccount.com
```

b. IAM Role & Admin

- **What it is:** IAM (Identity and Access Management) defines permissions. We need to grant Gmail's own internal service account permission to publish messages to our Pub/Sub topic.
- **Why we use it:** It's a fundamental security practice. By granting only the necessary Pub/Sub Publisher role, we ensure that nothing else can send messages to our topic.
- **Console:** Pub/Sub > Topics > select topic > Permissions > Add Principal.
- **gcloud Shell:**

```
gcloud pubsub topics add-iam-policy-binding
gmail-notification-topic \
  --member="serviceAccount:google-gmail-api@google.com" \
  --role="roles/pubsub.publisher"
```

c. Pub/Sub Topic & Subscription with Dead Letter Queue (DLQ)

- **What they are:**
 - **Topic:** A named channel where messages are sent (e.g., gmail-inbox-updates).
 - **Subscription:** A named endpoint that receives messages from a topic.
 - **DLQ (Dead-Letter Queue):** A separate topic where Pub/Sub sends messages that our application fails to process after several retries.
- **Why we use them:** This is the core of our real-time system. The DLQ is crucial for reliability; it prevents us from losing important email notifications if our webhook is down or returns an error.
- **Console:** Pub/Sub > Topics > Create Topic. Then Create Subscription from the topic page.
- **gcloud Shell:**

```
# Create the main topic
gcloud pubsub topics create gmail-notification-topic

# Create the Dead-Letter Queue topic
gcloud pubsub topics create gmail-notification-dlq-topic

# Create a DLQ Subscription
gcloud pubsub subscriptions create gmail-notification-dlq-sub \
  --topic=gmail-notification-dlq-topic
```

```
# Create the push subscription with the DLQ configured
gcloud pubsub subscriptions create gmail-notification-sub \
--topic=gmail-notification-topic \
--dead-letter-topic=gmail-notification-dlq-topic \
--max-delivery-attempts=3 \
--retry-policy=minBackoff=30s,maxBackoff=300s
```

d. Enable Gmail API & Give Access

- **What it is:** The process of activating the Gmail API in our Google Cloud project and configuring the OAuth consent screen that users will see.
- **Why we use it:** Our application cannot access any Gmail data until the API is enabled and a user has explicitly granted permission through the consent screen.
- **Console:** APIs & Services > Library > search for Gmail API and Enable. Then configure the OAuth consent screen.
- **gcloud Shell:**
gcloud services enable [gmail.googleapis.com](https://console.cloud.google.com/apis/library/gmail.googleapis.com) --project=PROJECT_ID
gcloud services enable [pubsub.googleapis.com](https://console.cloud.google.com/apis/library/pubsub.googleapis.com) --project=PROJECT_ID

8.2. LLM Implementation

a. Why LLM over Manual Mapping?

- **What it is:** Manual mapping uses simple keywords (e.g., if email contains "password", categorize as "Login Issue"). An LLM understands context, sentiment, and intent.
- **Why we use it:** LLMs are far more accurate. A customer might say "I can't get in" instead of "password reset." A keyword-based system would fail, but an LLM understands the intent and categorizes it correctly.

For more detail : [📖 Comparison: Manual Mapping vs. LLM for Email Categorization](#)

b. How to Install Specific Transformer Versions

- **What it is:** Installing specific, compatible versions of Python libraries.
- **Why we use it:** The machine learning ecosystem moves fast. Installing the latest versions of libraries like transformers and torch can lead to unexpected conflicts. Pinning to versions known to work together ensures stability and avoids

hard-to-debug errors.

```
# These versions are known to be stable and compatible
pip install transformers==4.30.2 torch==2.0.1 sentencepiece==0.1.99
safetensors==0.3.1
```

c. safetensors and sentencepiece

- **What they are:**
 - **safetensors:** A modern, secure file format for storing the large numerical weights of an LLM.
 - **sentencepiece:** A tokenizer that breaks down text into smaller, universal pieces that the model can process.
- **Why we need them:**
 - We use safetensors because it's faster and safer than the older pickle format, protecting us from potentially malicious code in model files.
 - We need sentencepiece because our chosen model is multilingual. It's a powerful tokenizer that works across many languages without needing separate logic for each one.

d. Why Zero-Shot Classification?

- **What it is:** A technique where a pre-trained model can classify text into categories it has never seen during its training. We provide the categories at the time of inference.
- **Why we use it:** We don't have a large dataset of thousands of labeled customer emails to fine-tune our own custom model from scratch. Zero-Shot Classification allows us to get started immediately with high accuracy, simply by providing a list of our desired categories (e.g., ["Payment", "Login", "Bug"]).

ref:

https://huggingface.co/docs/transformers/en/main_classes/pipelines#transformers.ZeroShotClassificationPipeline

e. Why the Hugging Face NLI Model?

- **What it is:** A large, powerful, multilingual model specifically trained for Natural Language Inference (NLI), which makes it excellent for Zero-Shot Classification.
- **Why we use it:** Its multilingual capability is a key advantage, allowing the system to handle emails in various languages automatically. Its large size provides higher accuracy than smaller models, which is crucial for a customer-facing application.

8.3. Backend Implementation (Simple Code)

a. & b. OAuth2 Start and Callback

File: customer_module/views/views_api_v1.py

```
from django.shortcuts import redirect
from django.conf import settings
from rest_framework.views import APIView
from rest_framework.response import Response
from google_auth_oauthlib.flow import Flow
from .models import GmailToken # Assume have this model

CLIENT_SECRETS_FILE = settings.GOOGLE_CREDENTIALS_FILE
SCOPES = ['https://www.googleapis.com/auth/gmail.modify']

class GoogleOAuthStartView(APIView):
    def get(self, request):
        flow = Flow.from_client_secrets_file(
            CLIENT_SECRETS_FILE,
            scopes=SCOPES,
            redirect_uri=request.build_absolute_uri('/api/oauth2/callback')
        )
        authorization_url, state = flow.authorization_url(access_type='offline', prompt='consent')
        request.session['state'] = state
        return redirect(authorization_url)

class GoogleOAuthCallbackView(APIView):
    def get(self, request):
        state = request.session['state']
        flow = Flow.from_client_secrets_file(
            CLIENT_SECRETS_FILE,
            scopes=SCOPES,
            state=state,
            redirect_uri=request.build_absolute_uri('/api/oauth2/callback')
        )
        flow.fetch_token(authorization_response=request.build_absolute_uri())
        credentials = flow.credentials

        user_info = flow.authorized_session().get('https://www.googleapis.com/oauth2/v3/userinfo').json()
```

```

user_email = user_info['email']

GmailToken.objects.update_or_create(
    user_email=user_email,
    defaults={'credentials': credentials.to_json()}
)
return Response({"message": "Authentication successful."})

```

c. & d. Register and Stop Gmail Watch

File: customer_module/views/views_api_v1.py

```

from googleapiclient.discovery import build
from google.oauth2.credentials import Credentials
from .models import GmailToken

SCOPES = ['https://www.googleapis.com/auth/gmail.modify']

def get_gmail_service(user_email):
    token_data = GmailToken.objects.get(user_email=user_email)
    creds =
    Credentials.from_authorized_user_info(json.loads(token_data.credentials),
    SCOPES)
    return build('gmail', 'v1', credentials=creds)

def register_watch_service(user_email):
    service = get_gmail_service(user_email)
    request = {
        'labelIds': ['INBOX'],
        'topicName': 'projects/PROJECT_ID/topics/gmail-notification-topic
    }
    return service.users().watch(userId='me', body=request).execute()

def stop_watch_service(user_email):
    service = get_gmail_service(user_email)
    return service.users().stop(userId='me').execute()

```

e. Pub/Sub Callback & Email Classification

File: customer_module/tasks.py/customer_related.py


```

import base64, json, email
from celery import shared_task
from transformers import pipeline
from .services import get_gmail_service
from .models import AutoReplyTemplate

classifier = pipeline("zero-shot-classification",
model="joeddav/xlm-roberta-large-xnli")

@shared_task
def process_email_task(user_email, message_id):
    service = get_gmail_service(user_email)
    msg_raw = service.users().messages().get(userId='me', id=message_id,
format='raw').execute()
    # Parse email body (simplified)
    # Sshould extract email_body from msg_raw['raw'] properly

    candidate_labels =
list(AutoReplyTemplate.objects.filter(is_active=True).values_list('category',
flat=True))
    if not candidate_labels:
        return

    result = classifier(email_body, candidate_labels)
    category, confidence = result['labels'][0], result['scores'][0]

    if confidence < 0.75:
        return # Do not reply if confidence is low (storing to our DB)

    try:
        template = AutoReplyTemplate.objects.get(category=category,
is_active=True)
        # Build and send reply Logic here
    except AutoReplyTemplate.DoesNotExist:
        return

```

File: customer_module/views/views_api_v1.py

```

from rest_framework.views import APIView
from rest_framework.response import Response
class PubSubWebhookView(APIView):
    def post(self, request):
        envelope = request.data
        message = envelope['message']

```

```
data_str = base64.b64decode(message['data']).decode('utf-8')
message_data = json.loads(data_str)
user_email = message_data['emailAddress']
# Example: Extract message_id from message_data if available
# message_id = message_data.get('messageId')
process_email_task.delay(user_email, message_id)
```

8.4 Example Result

Predicted Labels

['loan', 'registration', 'late_disbursement', 'forget_pin'] -> for example

These are the possible categories the model has been trained to predict. It attempts to assign one or more of these labels to the input sentence.

Email body

"Dari kemarin saya stuck di halaman tarik dana, saya ga bisa lanjut ketika sudah menentukan nominal pinjaman saya"


Result:

```
{
  "sequence": "Dari kemarin saya stuck di halaman tarik dana, saya ga bisa
lanjut ketika sudah menentukan nominal pinjaman saya",
  "labels": [
    "loan",
    "registration",
    "late_disbursement",
    "forget_pin"
  ],
  "scores": [
    0.9004806280136108,
    0.08501661568880081,
    0.012599742971360683,
    0.0019030317198485136
  ]
}
```

}

Confidence score

[0.9004806280136108, 0.08501661568880081, 0.012599742971360683, 0.0019030317198485136]

Label	Confidence Score
loan	0.900  (very high confidence)
registration	0.085
late_disbursement	0.012
forget_pin	0.0019

Interpretation

The model is **very confident (90%)** that this sentence is about a **loan**.

It is **less likely** to be about **registration, late_disbursement, or forget_pin**

8.5 The Improvement Area

Move from a zero-shot classification and static response system -> to a **context-aware, personalized response generator** powered by **RAG and fine-tuned LLMs**.

What is RAG?

Retrieval-Augmented Generation is a method that combines:

1. **Retriever:** Finds relevant documents (e.g. FAQ, SOP, past email responses data).

2. **Generator:** Uses the retrieved context + user input to generate a more accurate and personalized response.

Technical Requirements

1. Data Preparation Pipeline

We need structured and clean data from:

- **complaint_email_log** (body, predicted vs actual)
- SOP documents (PDFs, Docs)
- FAQ and knowledge base (Markdown, Notion, etc.)

Tools:

- **langchain** or **haystack** for text splitting & chunking
- **sentence-transformers** for embeddings
- Use **PyMuPDF**, **pdfminer**, **docx** for parsing raw documents

2. Vector Database (for RAG Retriever)

A retriever will need a **vector store** that supports semantic search, storing each chunk of our knowledge base as an embedded vector with metadata.

Options:

Tool	Description
FAISS	Fast local testing, easy setup (need to learn about KNN & ANN)
Weaviate	Open-source + scalable with REST API
Pinecone	Cloud-hosted, scalable
Qdrant	Fast, open-source, production-ready

PgVector	
-----------------	--

3. LLM Generator

RAG combines retrieved documents + prompt to generate responses.

Options:

Method	Model	Notes
OpenAI GPT-4	Via API	Fast start, pay-per-call
HuggingFace Model	tiiuae/falcon-7b , etc.	Run locally with GPUs
Custom fine-tuned	Using our labeled data	Best accuracy (takes effort)

4. RAG Frameworks

Use these tools to build the RAG pipeline:

Framework	Description
Langchain	Chains inputs → retriever → generator
Haystack	Modular NLP pipelines for search + gen

LlamaIndex	For document indexing, retrieval & prompt
-------------------	---

5. Infrastructure Needed

Component	Technology
Vector Store	FAISS / Weaviate / Qdrant
Embedding Model	all-MiniLM-L6-v2 or XLM-R
Generator Model	GPT-4 / Open-source HF models
Document Loader	langchain.document_loaders
Retriever Logic	similarity_search()
Django Backend	Serve API + queue RAG tasks

8.6 Suggested Roadmap

Phase 1 (Now): Zero-Shot + Predefine Template

Basic auto-reply based on LLM category.

Phase 2 (Soon): Human Review + Logging

- Store `actual_category`, `predicted_category`, `email_body`, `template_id` for logging data that can use for training data / fine tuning.
 - Build a feedback loop by building a CRM for agents that can fill the `actual_category`..
-

Phase 3: RAG Implementation (Contextual Reply)

- Ingest knowledge base (FAQ, SOP, past emails) into vector store.
 - Combine with LLM (e.g. GPT or Mistral) to create contextual replies.
-

Phase 4: Fine-Tuning

Once we collect a large dataset of:

- `email_body`
- `actual_category`
- `final_reply_text (template_id)`

We can fine-tune a small LLM to directly generate contextual responses without templates.

8.7 Predicted Cost

A. Gmail API

- Free quota: Up to 1,000,000 **messages.get**, 100,000 **messages.send** per day
- Your usage: 3,000 reads + 3,000 sends = 6,000 ops/day
Price: \$0 (well within free quota)

B. Google Pub/Sub

- **Free Quota:** 10 GB data + 10,000,000 messages/month
- **Usage:**
5,000 messages/day × 30 = **150,000 messages**
- **Price:** \$0.00 (well under free tier)

C. Cloud Compute

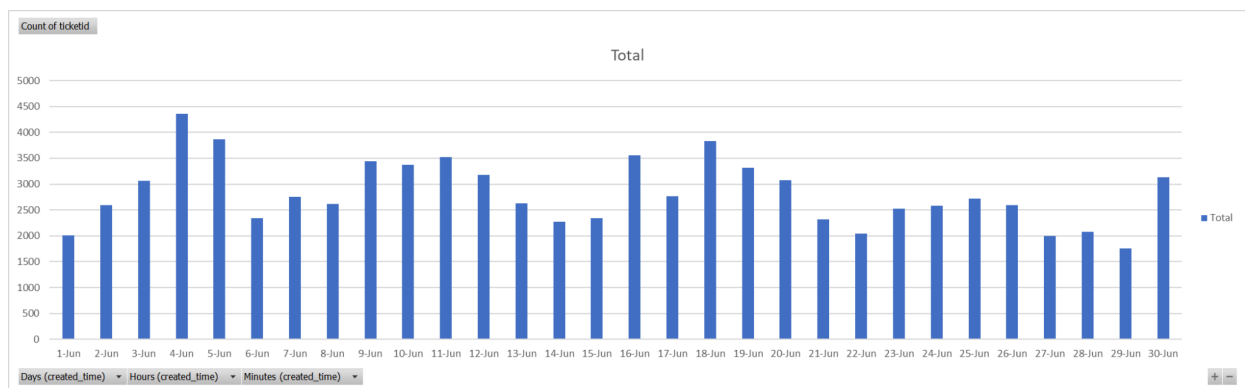
- To host the local model (`safetensors` ~1.1 GB), we need a compute instance with:
CPU: `n2-standard-4` (4 vCPU, 16 GB RAM)

- Using transformers and safetensors locally is **free**
- No Hugging Face API usage = no API cost
- **Price:** \$0.00

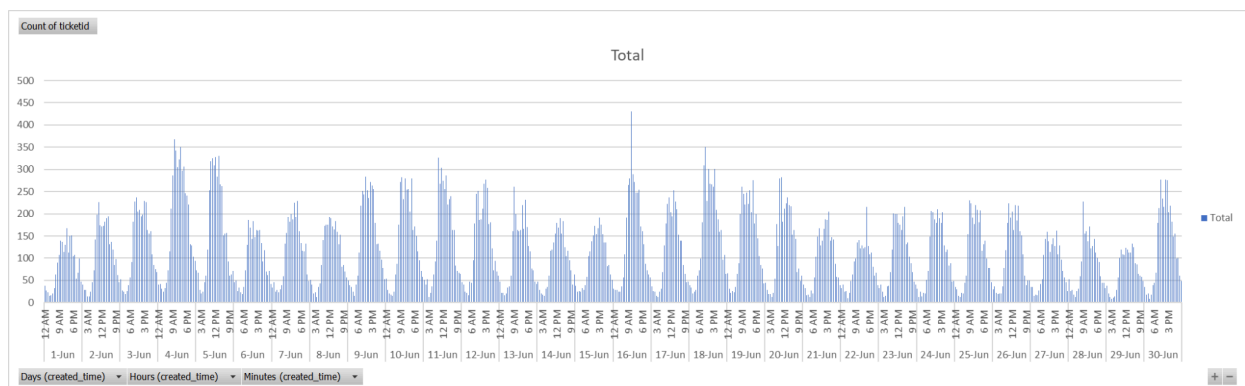
8.8 References

Range in June 2025

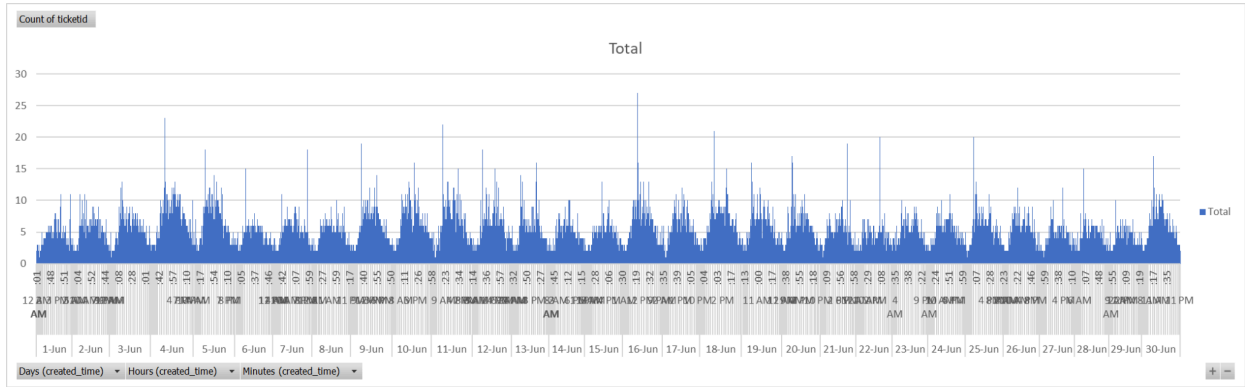
Email incoming per day (in June 2025)



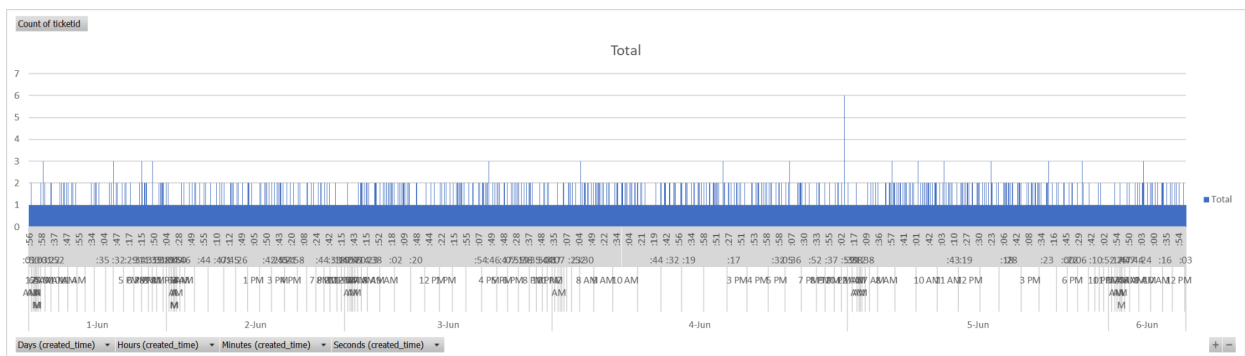
Email incoming per hour (in June 2025)



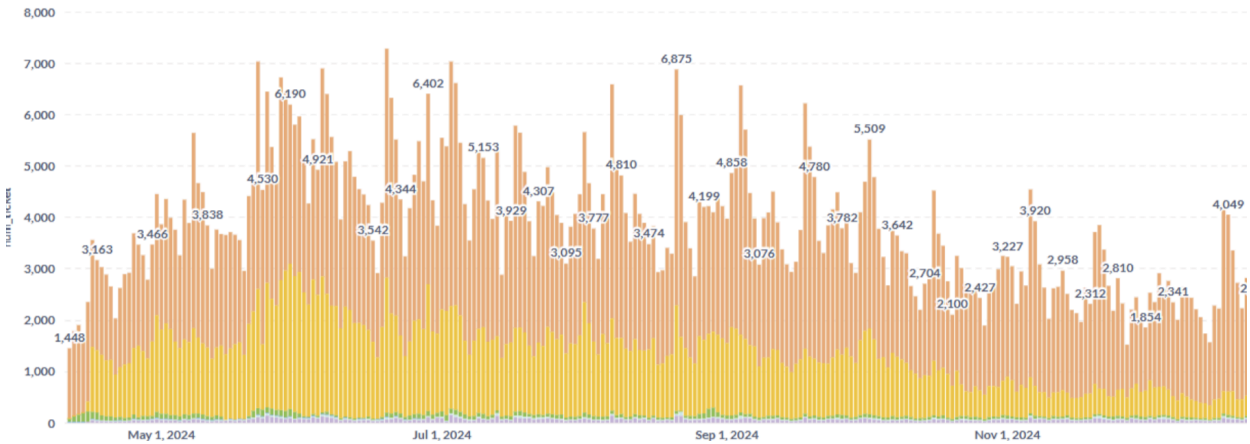
Email incoming per minute (in June 2025)



Email incoming per second (in June 2025)



Range between Jan and May 2025



Range between May and Dec 2024

