Voice Agent - Full Flask + Supabase + Twilio + Google Sheets + Atoms

This repository contains a complete, runnable **Flask** project implementing the SOP you shared: auto-dial contacts from Supabase, run a short scripted conversation (simple Twilio TTS version + production Atoms WebSocket streaming skeleton), append call outcomes to Google Sheets, and update Supabase. It includes background job processing (RQ + Redis), a WebSocket relay skeleton for Twilio Media Streams -> Atoms, and utility scripts.

File tree

```
voice-agent-project/
├─ README.md
├─ requirements.txt
├─ docker-compose.yml
├─ .env.example
├─ migrations/
│  └─ 001_create_contacts_to_call.sql
├─ app/
│  ├─ __init__.py
│  ├─ config.py
│  ├─ web.py                 # Flask app (HTTP endpoints)
│  ├─ ws_relay.py          # ASGI WebSocket relay for Twilio Media Streams ->
Atoms
│  ├─ jobs.py              # background job functions
│  ├─ worker.py            # RQ worker entry
│  ├─ sheets.py            # Google Sheets helper
│  ├─ supabase_client.py  # supabase helper
│  └─ atoms_client.py     # Atoms integration stubs
└─ deploy/
   └─ nginx.conf
```

> **Important**: replace sensitive values with your environment variables. Never commit service keys.

README (quick)

```
# Voice Agent Project

## Overview
Auto-dials contacts in Supabase, runs a short scripted conversation (Twilio
TTS or Atoms-powered), appends results to Google Sheets, updates Supabase,
tracks attempts and dispositions.

## Local setup
1. Copy `.env.example` to `.env` and set values.
2. Start Redis: `docker-compose up -d redis`
3. Start Flask server: `python -m app.web`
4. Start RQ worker: `python -m app.worker`
5. Expose public URL with `ngrok http 8000` and set `BASE_URL` env.
6. Configure Twilio webhooks for status callbacks and answer URLs.

## Files
See README in repository for full details.
```

---

requirements.txt

```
Flask==2.3.3
supabase-py==1.1.0
twilio==8.4.1
google-api-python-client==2.88.0
google-auth==2.22.0
rq==1.16.0
redis==4.7.0
python-dotenv==1.0.0
uvicorn==0.22.0
websockets==11.0.3
requests==2.31.0
pytz==2025.7
aiohttp==3.9.9
```

---

docker-compose.yml (Redis only)

```
version: '3.8'
services:
  redis:
    image: redis:7
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
volumes:
  redis-data:
```

.env.example

```
# Supabase
SUPABASE_URL=https://your-project.supabase.co
SUPABASE_SERVICE_KEY=your-service-role-key

# Twilio
TWILIO_ACCOUNT_SID=ACxxxxxxxxxxxxxxxxxxxx
TWILIO_AUTH_TOKEN=your_auth_token
TWILIO_FROM_NUMBER=+1234567890

# Google Sheets
GOOGLE_SERVICE_ACCOUNT_JSON_PATH=/path/to/service-account.json
GOOGLE_SHEET_ID=your_google_sheet_id

# App
BASE_URL=https://your-public-domain-or-ngrok
CALL_BATCH_SIZE=5
REDIS_URL=redis://localhost:6379/0

# Atoms
ATOMS_API_KEY=your_atoms_api_key
ATOMS_WS_URL=wss://atoms.smallest.ai/media

# Misc
FLASK_ENV=development
```

---

migrations/001_create_contacts_to_call.sql

```sql
CREATE TABLE IF NOT EXISTS public.contacts_to_call (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  name text,
  phone text,
  status text DEFAULT 'new',
  attempts integer DEFAULT 0,
  last_attempt_at timestamptz,
  metadata jsonb DEFAULT '{}'::jsonb
);

-- Index on metadata->call_sid would be helpful if you store call SID there
CREATE INDEX IF NOT EXISTS idx_contacts_metadata_call_sid ON
public.contacts_to_call((metadata->>'call_sid'));
```

---

app/config.py

```
import os
from dotenv import load_dotenv
load_dotenv()

class Config:
    SUPABASE_URL = os.environ['SUPABASE_URL']
    SUPABASE_KEY = os.environ['SUPABASE_SERVICE_KEY']
    TW_SID = os.environ['TWILIO_ACCOUNT_SID']
    TW_TOKEN = os.environ['TWILIO_AUTH_TOKEN']
    TW_FROM = os.environ['TWILIO_FROM_NUMBER']
    BASE_URL = os.environ.get('BASE_URL')
    SHEET_ID = os.environ['GOOGLE_SHEET_ID']
    GOOGLE_SA_JSON = os.environ['GOOGLE_SERVICE_ACCOUNT_JSON_PATH']
    CALL_BATCH_SIZE = int(os.environ.get('CALL_BATCH_SIZE', 5))
    REDIS_URL = os.environ.get('REDIS_URL', 'redis://localhost:6379/0')
    ATOMS_API_KEY = os.environ.get('ATOMS_API_KEY')
    ATOMS_WS_URL = os.environ.get('ATOMS_WS_URL')
```

---

app/supabase_client.py

```
from supabase import create_client
from .config import Config

supabase = create_client(Config.SUPABASE_URL, Config.SUPABASE_KEY)

# small helpers
def fetch_new_contacts(limit=None):
    q = supabase.table('contacts_to_call').select('*').eq('status',
'new').lt('attempts', 3)
    if limit:
        q = q.limit(limit)
    return q.execute().data or []

def mark_queued(ids):
    if not ids:
        return
    supabase.table('contacts_to_call').update({'status':'queued'}).in_('id',
ids).execute()

def update_contact(id_, payload):
    return supabase.table('contacts_to_call').update(payload).eq('id',
id_).execute()

def find_contact_by_call_sid(call_sid):
    # we created index to query metadata->>call_sid
    resp = supabase.table('contacts_to_call').select('*').filter("metadata->>call_sid", "=", call_sid).execute()
    items = resp.data or []
    return items[0] if items else None
```

---

app/sheets.py

```
from google.oauth2 import service_account
from googleapiclient.discovery import build
from .config import Config

def sheets_service():
    creds = service_account.Credentials.from_service_account_file(
        Config.GOOGLE_SA_JSON,
        scopes=["https://www.googleapis.com/auth/spreadsheets"]
    )
    return build('sheets', 'v4', credentials=creds)

def append_call_log_row(row_values):
    svc = sheets_service()
    body = {'values': [row_values]}
    svc.spreadsheets().values().append(
        spreadsheetId=Config.SHEET_ID,
        range='call_log!A1',
        valueInputOption='USER_ENTERED',
        body=body
    ).execute()
```

---

app/atoms_client.py

```
"""
Stubs for Atoms integration. The exact media protocol may differ — replace with Atoms-specific flow.

This file provides:
- `send_audio_to_atoms` : sends audio or chunks to Atoms and receives transcripts / decisions
- `start_session_with_atoms` : obtains a session id if needed
- `close_session` : clean up

You should replace placeholders with Atoms documentation calls (websocket or REST) you have in your account.
"""

import asyncio
```

```python
import json
from .config import Config

# Example async stub to forward audio frames via websocket to Atoms
async def atoms_relay_websocket(ws_uri, incoming_queue, outgoing_queue,
session_meta=None):
    """Connect to atoms websocket and forward frames.

    Args:
        ws_uri: Atoms websocket URL e.g. ws://atoms.smallest.ai/media
        incoming_queue: asyncio.Queue for audio frames from Twilio
        outgoing_queue: asyncio.Queue to receive TTS/audio frames or JSON
events from Atoms
    """
    # This is a stub. Replace with actual websocket code using `websockets`
or `aiohttp`.
    # Pseudocode:
    # async with websockets.connect(ws_uri, extra_headers={ 'Authorization':
f'Bearer {Config.ATOMS_API_KEY}' }) as ws:
    #   # send init message if required
    #   await ws.send(json.dumps({'type': 'session.start', 'meta':
session_meta}))
    #   while True:
    #       frame = await incoming_queue.get()
    #       await ws.send(frame)
    #       response = await ws.recv()
    #       await outgoing_queue.put(response)
    raise NotImplementedError("Please implement atoms_relay_websocket based
on Atoms API docs")
```

---

app/jobs.py

```python
import time
from datetime import datetime, timezone
from twilio.rest import Client as TwilioClient
from twilio.twiml.voice_response import VoiceResponse, Gather
from .config import Config
from .supabase_client import update_contact
from .sheets import append_call_log_row
from .supabase_client import supabase

twilio_client = TwilioClient(Config.TW_SID, Config.TW_TOKEN)

def place_call(contact):
    """Place an outbound call via Twilio and update Supabase with call_sid
and attempts."""
    answer_url = f"{Config.BASE_URL}/twiml/answer?contact_id={contact['id']}"
    callback_url = f"{Config.BASE_URL}/webhook/call-status"
    call = twilio_client.calls.create(
        to=contact['phone'],
        from_=Config.TW_FROM,
        url=answer_url,
        status_callback=callback_url,
        status_callback_event=['completed','failed','no-answer','busy'],
        status_callback_method='POST'
    )

    # increment attempts and store call_sid in metadata
    attempts = (contact.get('attempts') or 0) + 1
    md = contact.get('metadata') or {}
    md['call_sid'] = call.sid
    update_contact(contact['id'], {
        'status': 'dialed',
        'attempts': attempts,
        'last_attempt_at': datetime.now(timezone.utc).isoformat(),
        'metadata': md
    })
    return call.sid

def process_call_status(call_sid, call_status, duration, contact):
    """Process Twilio call status webhook and append to Google Sheet."""
    disposition = 'no_answer'
    if call_status == 'completed' and int(duration) > 0:
        disposition = 'connected'
    elif call_status in ('busy',):
        disposition = 'busy'
    elif call_status in ('failed','no-answer'):
        disposition = 'no_answer'

    # update supabase
    md = (contact.get('metadata') or {})
    md['last_status'] = call_status
    update_contact(contact['id'], {
        'status': 'completed' if disposition in
('connected','no_answer','busy') else 'failed',
        'metadata': md,
        'last_attempt_at': datetime.now(timezone.utc).isoformat()
    })

    # append to sheet
    row = [
        datetime.now(timezone.utc).isoformat(),
        call_sid,
        contact['id'],
        contact.get('name'),
        contact.get('phone'),
        disposition,
        f"Twilio status: {call_status}",
        '',
        duration
    ]
    append_call_log_row(row)
```

---

app/web.py (Flask app with endpoints)

```
from flask import Flask, request, Response, jsonify
from twilio.twiml.voice_response import VoiceResponse, Gather
from .config import Config
from .supabase_client import fetch_new_contacts, mark_queued, supabase,
find_contact_by_call_sid
from .jobs import place_call, process_call_status
from rq import Queue
from redis import Redis
from .config import Config
import json

app = Flask(__name__)

# RQ
redis_conn = Redis.from_url(Config.REDIS_URL)
q = Queue(connection=redis_conn)

@app.route('/start-batch', methods=['POST'])
def start_batch():
    contacts = fetch_new_contacts(limit=Config.CALL_BATCH_SIZE)
    ids = [c['id'] for c in contacts]
    mark_queued(ids)
    for c in contacts:
        q.enqueue(place_call, c)
    return jsonify({'queued': len(contacts)})

# TwiML initial answer (simple scripted flow)
@app.route('/twiml/answer', methods=['POST','GET'])
def twiml_answer():
    contact_id = request.values.get('contact_id')
    resp = supabase.table('contacts_to_call').select('*').eq('id',
contact_id).single().execute()
    contact = resp.data
    vr = VoiceResponse()
    vr.say('Hello. This is a short call from Example Company.',
voice='alice', language='en-US')
    vr.pause(length=0.4)
    vr.say(f"Am I speaking with
{contact.get('name')}? Press 1 for yes, 2 for no.")
    g = Gather(num_digits=1, action=f"{Config.BASE_URL}/twiml/gather_verify?
contact_id={contact_id}", method='POST', timeout=5)
    vr.append(g)
    vr.say('We did not receive an answer. Goodbye.')
    vr.hangup()
    return Response(str(vr), mimetype='application/xml')

@app.route('/twiml/gather_verify', methods=['POST'])
def gather_verify():
    digit = request.values.get('Digits')
    contact_id = request.values.get('contact_id')
    vr = VoiceResponse()
    if digit == '1':
        vr.say('Great, thank you. I have two quick questions.',
voice='alice')

vr.say('Where do you work? You can record your answer after the beep. Press #
when done.')
        vr.record(max_length=12, finish_on_key='#',
action=f"{Config.BASE_URL}/twiml/record1?contact_id={contact_id}")
        return Response(str(vr), mimetype='application/xml')
    else:
        vr.say('Sorry we reached the wrong person. Thank you. Goodbye.',
voice='alice')
        vr.hangup()
        return Response(str(vr), mimetype='application/xml')

@app.route('/twiml/record1', methods=['POST'])
def record1():
    recording_url = request.values.get('RecordingUrl')
    contact_id = request.values.get('contact_id')
    # store recording url into metadata
    resp = supabase.table('contacts_to_call').select('*').eq('id',
contact_id).single().execute()
    contact = resp.data
    md = contact.get('metadata') or {}
    md['recording_1'] = recording_url
    supabase.table('contacts_to_call').update({'metadata': md}).eq('id',
contact_id).execute()

    vr = VoiceResponse()

vr.say('Thank you. One last question: what do you do? Please record after the
beep and press # when done.')
    vr.record(max_length=12, finish_on_key='#', action=f"{Config.BASE_URL}/
twiml/record2?contact_id={contact_id}")
    return Response(str(vr), mimetype='application/xml')

@app.route('/twiml/record2', methods=['POST'])
def record2():
    recording_url = request.values.get('RecordingUrl')
    contact_id = request.values.get('contact_id')
    resp = supabase.table('contacts_to_call').select('*').eq('id',
contact_id).single().execute()
```

```
    contact = resp.data
    md = contact.get('metadata') or {}
    md['recording_2'] = recording_url
    supabase.table('contacts_to_call').update({'metadata': md}).eq('id',
contact_id).execute()

    vr = VoiceResponse()
    vr.say('Thanks for your time. Good bye.', voice='alice')
    vr.hangup()
    return Response(str(vr), mimetype='application/xml')

# Twilio status callback
@app.route('/webhook/call-status', methods=['POST'])
def call_status():
    call_sid = request.values.get('CallSid')
    call_status = request.values.get('CallStatus')
    duration = request.values.get('CallDuration') or '0'

    contact = find_contact_by_call_sid(call_sid)
    if contact:
        q.enqueue(process_call_status, call_sid, call_status, duration,
contact)
        return ('', 204)
    else:
        # fallback: search by phone
        return ('', 204)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000, debug=True)
```

app/worker.py

```
# run with: python -m app.worker
from rq import Worker, Queue, Connection
from redis import Redis
from .config import Config

listen = ['default']
redis_conn = Redis.from_url(Config.REDIS_URL)
if __name__ == '__main__':
    with Connection(redis_conn):
        worker = Worker(map(Queue, listen))
        worker.work()
```

app/ws_relay.py (ASGI WebSocket relay skeleton)

```
"""
ASGI WebSocket relay that accepts Twilio Media Streams websocket connections
and forwards audio to Atoms websocket.

This is a skeleton and requires filling in details from Atoms docs.

Run with: uvicorn app.ws_relay:app --host 0.0.0.0 --port 9000

Twilio will call your TwiML answer that connects to a <Stream url="wss://
yourhost:9000/stream?contact_id=...">.

Flow:
 - Twilio -> this WS endpoint (receives base64 pcm frames as JSON)
 - Relay audio frames to Atoms websocket using
atoms_client.atoms_relay_websocket
 - Forward TTS frames from Atoms back to Twilio
 - Listen for final event/summary from Atoms and persist to Supabase & Google
Sheets
"""

import asyncio
import json
import logging
from websockets import serve
from .config import Config
from .supabase_client import update_contact
from .sheets import append_call_log_row
from .atoms_client import atoms_relay_websocket

logging.basicConfig(level=logging.INFO)

async def handle_twilio_ws(websocket, path):
    """Handle Twilio Media Stream connection.

    Twilio sends JSON messages with event types like 'start','media','stop'.
Media frames are base64 pcm (audio) in `media.payload`.
    """
    params = dict()
    # extract query params from path if present
    # path may be like: /?contact_id=xxx&call_id=YYY
    try:
        qstr = path.split('?', 1)[1]
        for kv in qstr.split('&'):
            k,v = kv.split('=',1)
            params[k]=v
    except Exception:
        pass
```

```python
    contact_id = params.get('contact_id')
    call_id = params.get('call_id')

    incoming_queue = asyncio.Queue()
    outgoing_queue = asyncio.Queue()

    # spawn atoms relay task
    atoms_task =
asyncio.create_task(atoms_relay_websocket(Config.ATOMS_WS_URL,
incoming_queue, outgoing_queue, {'contact_id': contact_id, 'call_id':
call_id}))

    try:
        async for message in websocket:
            # Twilio messages are JSON strings
            obj = json.loads(message)
            event = obj.get('event')
            if event == 'start':
                logging.info('Stream started')
            elif event == 'media':
                # media.payload is base64 audio chunk
                media = obj.get('media', {})
                payload = media.get('payload')
                # push into incoming queue to send to Atoms
                await incoming_queue.put(payload)
            elif event == 'stop':
                logging.info('Stream stopped')
                break
            # check outgoing queue for atoms responses and send back to
Twilio
            while not outgoing_queue.empty():
                out = await outgoing_queue.get()
                # out should be JSON serializable
                await websocket.send(json.dumps(out))
    finally:
        atoms_task.cancel()
        # optionally collect summary from atoms and persist
        # example: append_call_log_row([...])

async def main():
    async with serve(handle_twilio_ws, '0.0.0.0', 9000):
        await asyncio.Future()

# For running with `python -m app.ws_relay`
if __name__ == '__main__':
    asyncio.run(main())
```

Notes on testing & deployment

- Use ngrok to expose `BASE_URL` for Twilio: `ngrok http 8000`. Set Twilio webhook `url` for the outgoing phone number or call creation to point to the TwiML endpoints.
- For media-streams, Twilio expects an HTTPS wss endpoint (public) reachable by Twilio. Use a server with a valid TLS cert or use a tool like `ngrok tcp` or paid tunnel.
- When moving to Atoms streaming, implement `atoms_relay_websocket` by following Atoms websocket API: open a session, forward chunks, consume JSON events (transcripts, final decisions), forward TTS chunks back to Twilio.

Explanations & recommendations

- **Simple flow** (the provided Flask TwiML endpoints) is quick to implement and test. It collects recorded answers and stores recording URLs for later transcription.
- **Production flow** uses Twilio Media Streams + Atoms to run a noisy real-time ASR+LLM agent. That requires more infra (ASGI server, websockets, concurrency). The skeleton `ws_relay.py` is the starting point.
- Use `rq` to decouple dialing (which may be rate-limited) from the main web thread.
- Use a secure secret manager for credentials.

If you want, I can now: - provide a single `docker-compose` to run the Flask app + redis + worker + uvicorn (and show how to wire them), or - implement the `atoms_relay_websocket` with a concrete example if you paste the Atoms media/websocket docs (or allow me to fetch them).