

Backend Architecture

Pet Love Song Generator — Pets at Home Valentine’s Day 2026

February 2026

Contents

Executive Summary	1
System Architecture Overview	2
Request Lifecycle	3
Celery Task Chain Architecture	5
Queue Topology & Concurrency Model	6
Cloud Run Video Offloading	8
Resilience & Failure Handling	10
Data Model	14
Infrastructure (Docker Compose)	16
Monitoring & Observability	17
Storage Architecture (GCS)	18
Reusability Guide	19
Appendix: Configuration Reference	20

Stack: FastAPI | Celery | PostgreSQL 15 | Redis 7 | Google Cloud Storage | Cloud Run

Executive Summary

The Pet Love Song Generator is a campaign microsite that lets users submit their pet’s name, photo, and a music vibe preference to receive an AI-generated love song rendered as a vinyl-record music video. The backend receives form submissions, dispatches them to an external AI music generation API (Sonauto), processes webhook callbacks, downloads the generated audio, and creates a composited video — all asynchronously.

The key architectural decision is **offloading CPU-intensive video rendering to Google Cloud Run**. The main backend handles API serving, database operations, and task orchestration, while each video render gets its own ephemeral Cloud Run container instance. This separation means that a burst of 100 simultaneous video renders doesn’t compete with API request handling or webhook processing on the main server. Cloud Run’s auto-scaling matches the bursty traffic pattern of a marketing campaign (peaks during social shares, lulls overnight), and its scale-to-zero capability keeps costs proportional to actual usage.

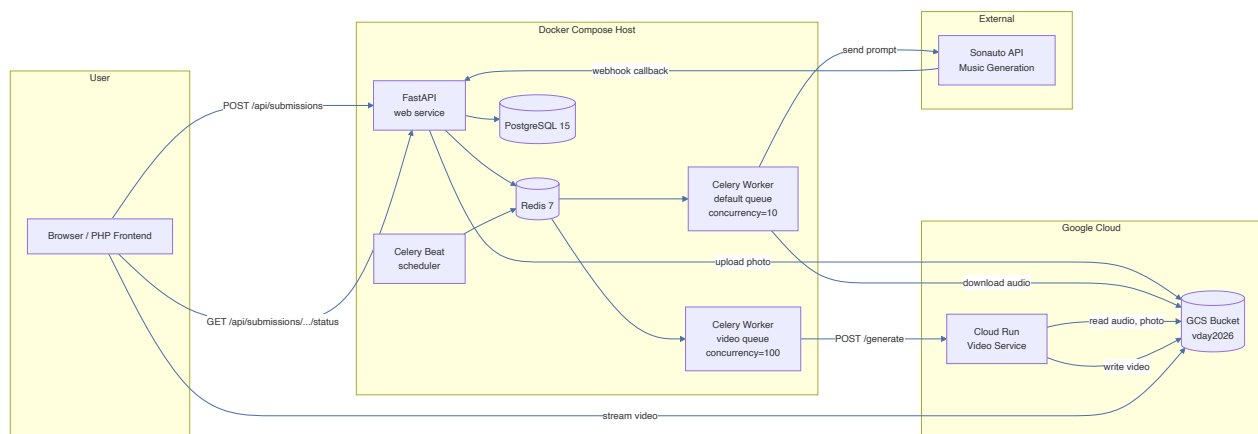
The system was designed to handle campaign-scale bursts — hundreds of concurrent submissions arriving within minutes of a social media push — while remaining cost-efficient during quiet periods. Resilience patterns (retry strategies, two-layer timeouts, fail-fast startup checks) ensure that transient failures in external APIs or infrastructure don't silently lose user submissions.

System Architecture Overview

The application runs as six Docker Compose services on a single host, plus a Cloud Run satellite service for video rendering:

Service	Role	Image
web	FastAPI application server	Custom Dockerfile
celery_worker_default	Default task worker (API calls, audio downloads, DB ops)	Custom Dockerfile
celery_worker_video	Video dispatch worker (HTTP calls to Cloud Run)	Custom Dockerfile
celery_beat	Periodic task scheduler	Custom Dockerfile
db	PostgreSQL 15 database	postgres:15
redis	Message broker + result backend + cache	redis:7-alpine

The Cloud Run video service runs independently, sharing only the GCS bucket as a data plane.



System architecture: Docker Compose services and external dependencies

Request Lifecycle

A submission moves through three distinct phases: **Submission**, **Generation**, and **Delivery**.

Phase 1: Submission

1. User fills out the form (pet name, owner name, pet type, music vibe, optional photo)
2. Browser sends POST `/api/submissions` with form data and base64-encoded cropped image
3. FastAPI validates input, runs server-side profanity check on pet/owner names
4. If a photo is provided, it's decoded from base64 and uploaded to GCS (`uploads/{session_id}.jpg`)
5. A `Submission` record is created in PostgreSQL with status pending
6. A `send_to_sonauto` Celery task is dispatched immediately
7. Response returns `session_id` to the browser for polling

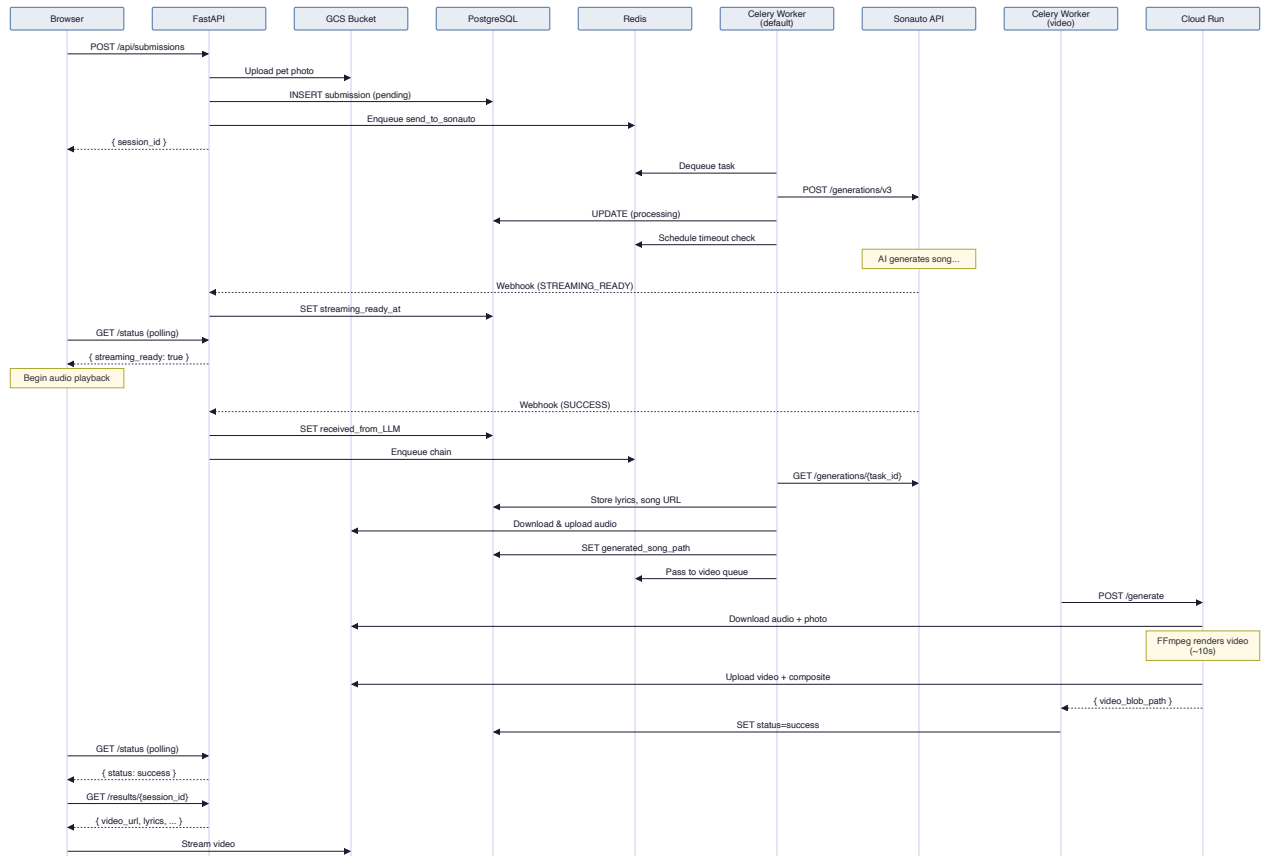
Phase 2: Generation

8. The `send_to_sonauto` task constructs a prompt from pet-specific templates and sends it to the Sonauto API
9. The submission status moves to `processing` and a per-submission timeout task is scheduled
10. Sonauto generates the song asynchronously and sends webhook callbacks:
 - `GENERATING_STREAMING_READY` — the audio stream is available for early playback (the frontend can start playing before the full song is done)
 - `SUCCESS` — full generation complete, triggers the post-webhook processing chain
 - `FAILURE` — marks the submission as failed
11. On `SUCCESS`, a Celery chain is triggered: `fetch_generation_details -> download_audio -> create_video`
12. The chain fetches song details from the Sonauto API, downloads the MP3 to GCS, then dispatches video creation
13. Video creation sends a request to Cloud Run, which renders the vinyl record animation using FFmpeg and uploads the result to GCS

Phase 3: Delivery

14. The frontend polls GET `/api/submissions/{session_id}/status` on an interval
15. Once status is success, the frontend redirects to the results page
16. The results page fetches GET `/api/results/{session_id}` which returns the video URL, lyrics, record image, and metadata
17. The video is served directly from GCS public URLs; downloads use signed URLs with `Content-Disposition: attachment`

Streaming-ready optimization: When the `GENERATING_STREAMING_READY` webhook arrives, the status endpoint includes `streaming_ready: true` and the Sonauto `task_id`. The frontend can begin audio playback via Sonauto's streaming endpoint while the full song and video are still being generated. This reduces perceived wait time significantly.



Request lifecycle: submission through delivery with streaming-ready optimization

Celery Task Chain Architecture

After the Sonauto webhook signals SUCCESS, a three-task Celery chain processes the result. Each task receives a dict from its predecessor and returns a dict consumed by the next.

Chain: `fetch_generation_details` -> `download_audio` -> `create_video`

Task 1 — `fetch_generation_details` - Calls GET `/generations/{task_id}` on the Sonauto API - Extracts song URL, lyrics, and status from the response - Stores the full API response, lyrics, and song URL in the database - Returns: `{ session_id, song_url, lyrics }`

Task 2 — `download_audio` - Downloads the MP3 from the Sonauto CDN URL - Uploads it to GCS at `audio/{session_id}.mp3` - Updates the database with the blob path - Returns: `{ session_id, audio_blob_path, lyrics }`

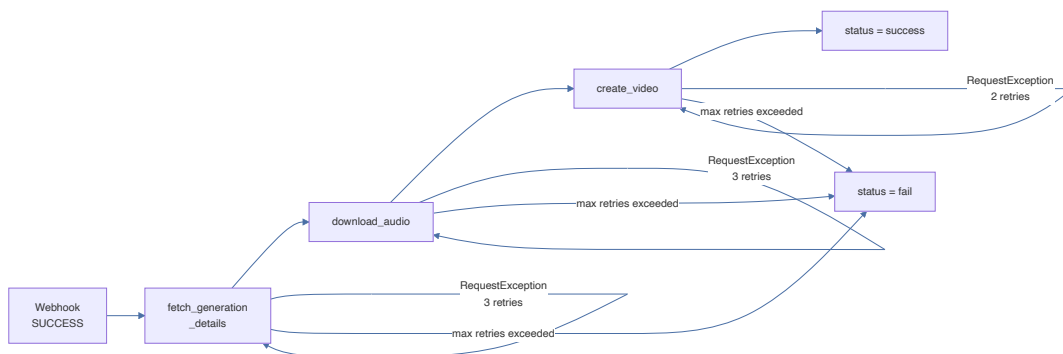
Task 3 — `create_video` - Routes to the video queue (separate worker pool) - If `CLOUD_RUN_VIDEO_URL` is configured: dispatches to Cloud Run via HTTP POST - Otherwise: falls back to local FFmpeg rendering (development mode) - Marks the submission as success or fail based on the outcome - Returns: `{ session_id, video_blob_path }`

Retry Configuration

All three chain tasks use the same retry pattern:

```
@shared_task(
    autoretry_for=(requests.RequestException,),
    retry_backoff=True,           # Exponential backoff
    retry_backoff_max=60,        # Cap at 60 seconds
    retry_kwargs={"max_retries": 3},
)
```

Additionally, `send_to_sonauto` tracks retries at the application level via a `retry_count` column. If the count reaches `MAX_RETRIES` (default: 3), the submission is marked as fail even before Celery exhausts its own retry budget — preventing indefinite requeuing.



Celery task chain with retry paths and failure handling

Queue Topology & Concurrency Model

This is the most critical load-handling pattern in the system. Two separate Celery queues with independent worker pools prevent video dispatch tasks from starving API-calling tasks during traffic bursts.

Why Queue Separation Matters

Video generation takes approximately 10 seconds per submission. During a traffic burst — say, 200 submissions arriving within a few minutes after a social media push — the system needs to:

1. Continue accepting and acknowledging Sonauto API calls (`send_to_sonauto`)
2. Continue processing webhook callbacks (the chain's first two tasks)
3. Dispatch up to 100 video renders in parallel to Cloud Run

Without queue separation, all these tasks would compete for the same worker slots. If the default worker had 10 concurrency slots, only 10 video dispatches could run at once — and while those slots are occupied waiting for Cloud Run HTTP responses (~10s each), no API calls or audio downloads could proceed. The entire pipeline would bottleneck.

Queue Configuration

“celery” queue (default) — concurrency: 10 - `send_to_sonauto` — outbound API calls to Sonauto - `fetch_generation_details` — fetch song metadata from Sonauto - `download_audio` — download MP3 from CDN, upload to GCS - `check_submission_timeout` — per-submission timeout checks - `check_timeouts` — periodic safety-net sweep - `cleanup_old_files` — daily file retention cleanup

These tasks involve API calls, file downloads, and database operations. Concurrency of 10 is sufficient because each task completes in seconds, and the external APIs have their own rate limits.

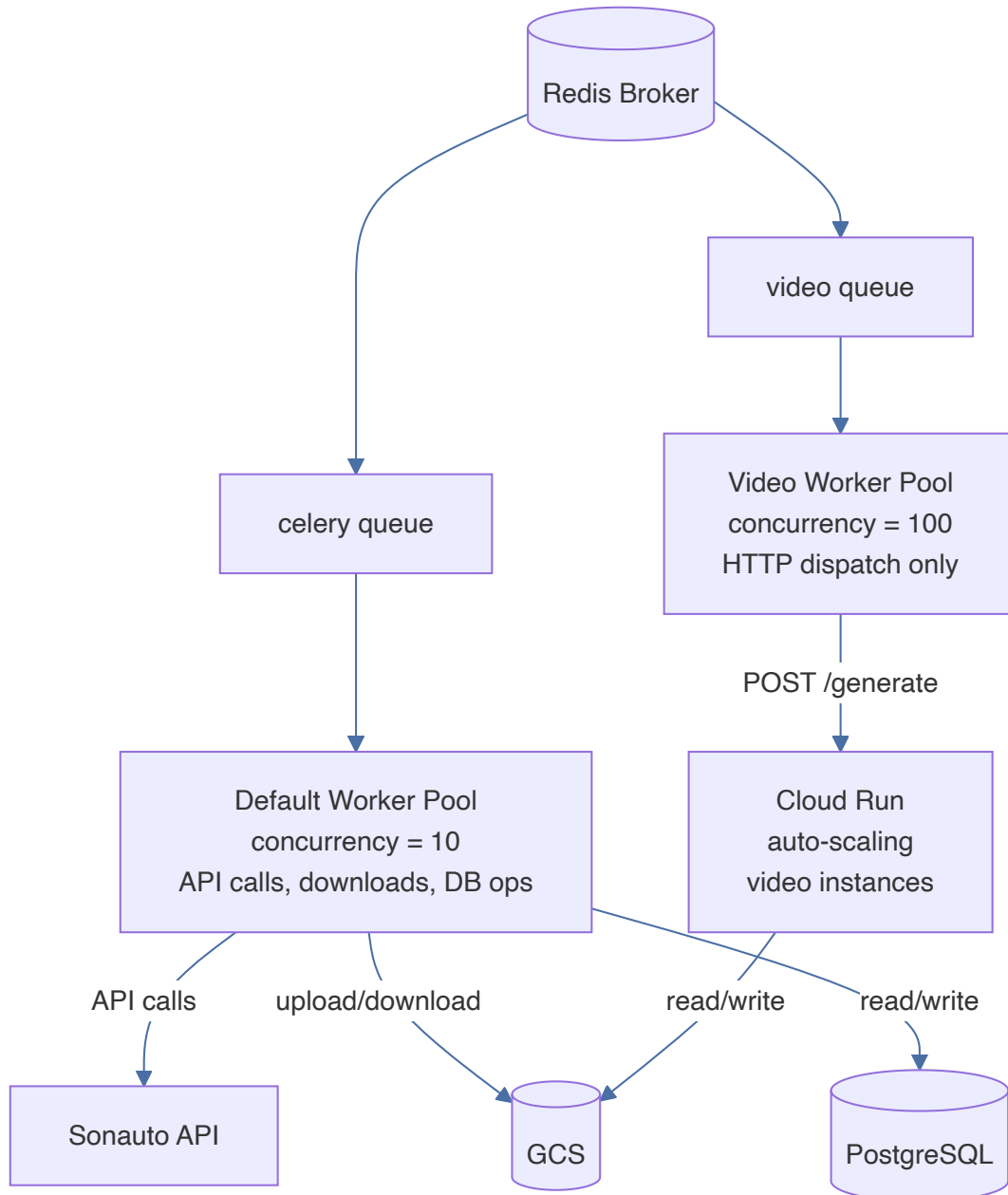
“video” queue — concurrency: 100 - `create_video` — HTTP dispatch to Cloud Run (or local FFmpeg fallback)

This worker does almost no local work. It sends an HTTP POST to Cloud Run and waits for the response. The actual CPU work happens on Cloud Run. With 100 concurrency, the system can have 100 Cloud Run instances rendering videos simultaneously. The high concurrency is safe because the worker is purely I/O-bound — it's just holding open HTTP connections.

Worker-Level Settings

```
# celery_app.py
"task_acks_late": True,
"worker_prefetch_multiplier": 1,
"task_routes": {
    "tasks.workers.create_video": {"queue": "video"},
},
"task_default_queue": "celery",
```

- **worker_prefetch_multiplier=1:** Each worker only grabs one task at a time from the broker. This ensures fair distribution across workers — without it, a single worker could prefetch a batch of tasks and create an uneven load.
- **task_acks_late=True:** Tasks are acknowledged only after completion, not when received. If a worker crashes mid-task, the message returns to the queue and another worker picks it up. This is the crash-recovery mechanism.



Queue topology: Redis broker feeding two independent worker pools

Cloud Run Video Offloading

Why Cloud Run?

FFmpeg video rendering is CPU-intensive. Each video takes approximately 10 seconds to render: generating ~45 frames for one rotation cycle, compositing four image layers per frame (background, pet photo, vinyl record, needle), then encoding to H.264. On the main server, running even 5 concurrent FFmpeg processes would consume significant CPU and memory, competing with FastAPI request handling and Celery task execution.

Cloud Run solves this in three ways:

1. **Horizontal auto-scaling:** Each video render request gets its own container instance. Cloud Run provisions instances on demand, so 100 simultaneous renders don't impact each other. There's no shared CPU to contend for.
2. **Isolation from the main backend:** The Celery video worker's job is reduced to a single HTTP POST — it doesn't consume CPU for rendering. The main server remains responsive for API requests, webhook processing, and database operations regardless of video rendering load.
3. **Scale-to-zero economics:** Campaign traffic follows a burst pattern (social shares trigger waves of submissions, then activity drops). Cloud Run scales to zero when idle, so there's no cost during lulls. This is far more cost-efficient than provisioning dedicated compute for peak load.

Data Flow

GCS serves as the shared data plane between the main backend and Cloud Run. There is no direct file transfer between services:

1. The Celery worker sends { `session_id`, `audio_blob_path`, `photo_blob_path` } to Cloud Run
2. Cloud Run downloads the audio and photo from GCS
3. Cloud Run runs FFmpeg to generate the video (720x1280px MP4, 15fps, vinyl record animation)
4. Cloud Run uploads the video and composite record image to GCS
5. Cloud Run returns { `video_blob_path` } to the Celery worker
6. The Celery worker updates the database with the video path and marks the submission as success

Configuration

```
CLOUD_RUN_VIDEO_URL=https://video-generator-xxx.a.run.app
VIDEO_SERVICE_API_KEY=shared-secret-for-auth
CLOUD_RUN_VIDEO_TIMEOUT=300 # 5 minutes safety ceiling
```

The 300-second timeout is a safety ceiling — videos normally render in ~10 seconds, but the timeout accounts for cold starts, GCS download time, and edge cases with very long audio tracks.

Local Development Fallback

When `CLOUD_RUN_VIDEO_URL` is not set, the `create_video` task falls back to local FFmpeg rendering. This allows development and testing without Cloud Run infrastructure. The local path downloads files from GCS to temp files, runs the video generator, uploads results, and cleans up.

Resilience & Failure Handling

8.1 Retry Strategy

All API-calling tasks use Celery's built-in exponential backoff retry:

- **Trigger:** `autoretry_for=(requests.RequestException,)` — any HTTP error triggers a retry
- **Backoff:** Exponential with `retry_backoff=True`, capped at `retry_backoff_max=60` seconds
- **Max retries:** 3 for API/download tasks, 2 for video creation
- **Application-level tracking:** The `send_to_sonauto` task also increments a `retry_count` column on the submission. If this count reaches `MAX_RETRIES` (configurable, default 3), the submission is marked as fail independently of Celery's retry mechanism. This prevents scenarios where Celery's retry counter resets (e.g., after a worker restart) from causing infinite retries.

8.2 Two-Layer Timeout Strategy

Submissions that are sent to Sonauto but never receive a webhook callback need to be detected and marked as failed. The system uses two complementary timeout mechanisms:

Layer 1 — Event-based per-submission countdown (primary)

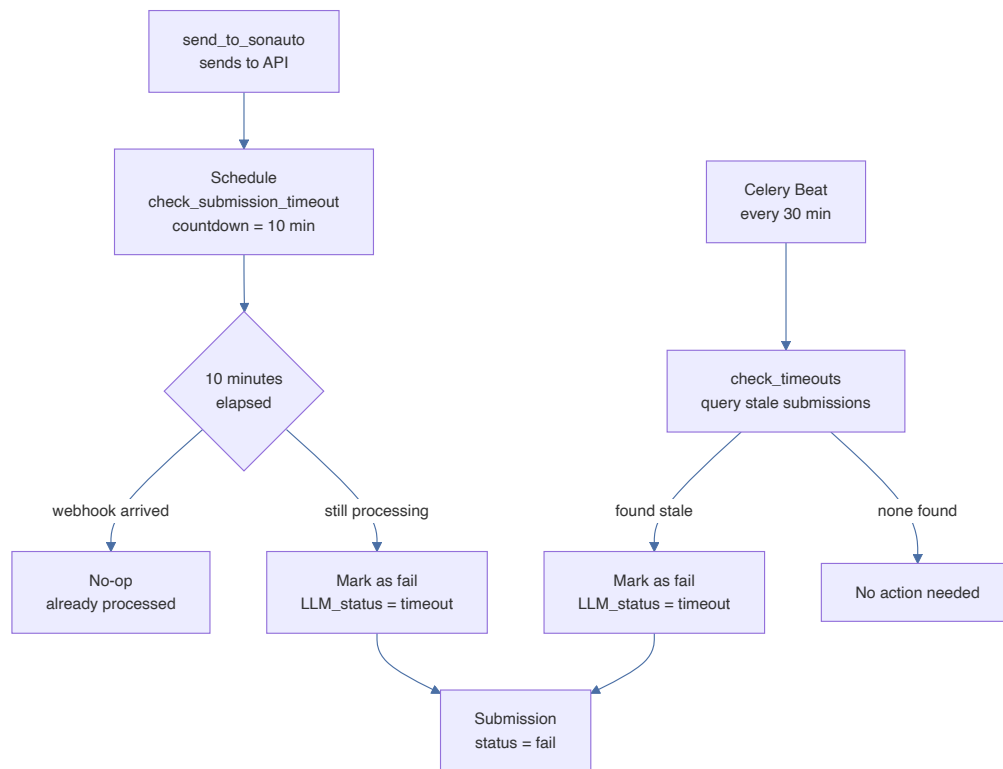
When `send_to_sonauto` successfully calls the Sonauto API, it immediately schedules a `check_submission_timeout` task with a countdown equal to `WEBHOOK_TIMEOUT_MINUTES` (default: 10 minutes):

```
check_submission_timeout.apply_async(
    args=[session_id],
    countdown=settings.WEBHOOK_TIMEOUT_MINUTES * 60,
)
```

This task fires at the exact timeout moment for each submission. If the webhook has already arrived, it's a no-op. If the submission is still in processing state, it's marked as fail with `LLM_status = "timeout"`.

Layer 2 — Periodic safety-net sweep (fallback)

Celery Beat runs `check_timeouts` every 30 minutes. This queries for any submissions where `sent_to_LLM` is older than the timeout threshold and `received_from_LLM` is still NULL. This catches edge cases where the per-submission timeout task itself failed to execute (e.g., the worker restarted, Redis lost the message).



Two-layer timeout strategy: event-based countdown and periodic sweep

8.3 Fail-Fast Startup

The FastAPI application verifies GCS connectivity on startup:

```
@app.on_event("startup")
async def startup_event():
    from app.services import storage
    storage.check_connectivity() # Raises if bucket inaccessible
```

If the GCS bucket is inaccessible (wrong credentials, bucket doesn't exist, network issue), the application refuses to start. This prevents a scenario where the app accepts submissions but silently fails to store images, leading to broken submissions that can never produce videos.

8.4 Health Monitoring

The `/api/health` endpoint performs three checks:

Check	Method	Unhealthy if
Redis	<code>redis_client.ping()</code>	Connection refused or timeout
PostgreSQL	<code>SELECT 1</code> via SQLAlchemy	Connection refused or query fails
Celery workers	<code>celery_app.control.inspect().active_tasks()</code>	No active workers detected

If any check fails, the endpoint returns HTTP 503. Load balancers use this to route traffic away from unhealthy instances.

8.5 Content Safety

Two layers of content filtering protect against inappropriate submissions:

Text filtering (client + server): - Client-side: JavaScript profanity check provides instant feedback
- Server-side: The `contains_profanity()` function uses the `better-profanity` library plus a custom banned-words list. This is the authoritative check — client-side filtering can be bypassed.

Image classification (Gemini AI): - Configurable via `IMAGE_SAFETY_METHOD` (default: `gemini`, alternative: `nudenet`) - Gemini receives the image with a strict classification prompt that only accepts photos with a pet/animal as the primary subject - **Fail-closed on content filter:** If Gemini's own safety filter blocks the response, the image is rejected - **Fail-open on network error:** If Gemini is unreachable (network error, auth failure), the image is allowed through — availability is prioritized over a rare edge case

8.6 Rate Limiting

Each user is identified by a `cookie_id` (generated on first submission, stored in `localStorage`). The server enforces a maximum of `FORM_SUBMIT_RETRY` (default: 10) submissions per `cookie_id` by counting existing submissions in the database. The client also enforces this limit for immediate feedback, but the server-side check is authoritative.

8.7 Database Connection Pooling

```
engine = create_engine(
    settings.DATABASE_URL,
    pool_pre_ping=True,      # Verify connections before use
    pool_size=10,           # Base pool size
    max_overflow=20,        # Additional connections under load
)
```

- **pool_pre_ping=True:** Before using a pooled connection, SQLAlchemy sends a lightweight ping. If the connection is stale (database restarted, network blip), it's discarded and a fresh one is created. This prevents "connection reset" errors after idle periods.
- **pool_size=10:** Maintains 10 persistent connections. Sufficient for the FastAPI server and Celery workers under normal load.
- **max_overflow=20:** Up to 20 additional connections can be created during traffic bursts, for a total of 30 concurrent connections. These overflow connections are closed after use.

8.8 Data Retention

A Celery Beat task runs daily at 3:00 AM UTC:

1. Queries submissions older than `FILE_RETENTION_DAYS` (default: 30 days)
2. Deletes associated files from GCS: uploaded photo, generated audio, generated video, and composite record image

3. Logs the count of deleted files and processed submissions

The database records themselves are not deleted — only the associated files. This preserves the audit trail while reclaiming storage.

Data Model

The system uses a single submissions table. Fields are grouped by purpose:

Identity & Tracking

Column	Type	Purpose
session_id	String(255) PK	CUID2-generated unique identifier
cookie_id	String(255) indexed	Client-side user identifier for rate limiting
created_at	DateTime	Submission timestamp

Form Data

Column	Type	Purpose
owner_name	String(100)	Pet owner's name
pet_name	String(100)	Pet's name
pet_type	String(50)	Pet species (dog, cat, etc.)
music_vibe	String(50)	Selected music style
photo_path	String(500)	GCS blob path to uploaded pet photo

Processing State

Column	Type	Purpose
entry_status	String(50)	Current status: pending -> processing -> success / fail
retry_count	Integer	Application-level retry counter for API calls

External API Tracking

Column	Type	Purpose
sent_to_LLM	DateTime	When the prompt was sent to Sonauto
LLM_task_id	String(255) indexed	Sonauto's task identifier for webhook correlation
received_from_LLM	DateTime	When the webhook callback was received
LLM_response	Text	Primary song URL from Sonauto
LLM_full_response	Text	Full JSON response for debugging
LLM_status	String(50)	Sonauto-specific status (success, fail, timeout)

Generated Content

Column	Type	Purpose
lyrics	Text	AI-generated song lyrics
generated_song_path	Text	GCS blob path to MP3 audio
generated_video_path	String(500)	GCS blob path to MP4 video

Timing & Streaming

Column	Type	Purpose
streaming_ready_at	DateTime	When Sonauto's streaming endpoint became available
video_creation_start	DateTime	When video rendering began
video_creation_end	DateTime	When video rendering completed

Indexes

- `ix_submissions_cookie_id` on `cookie_id` — fast rate-limit lookups
- `ix_submissions_llm_task_id` on `LLM_task_id` — fast webhook correlation by Sonauto task ID

Infrastructure (Docker Compose)

Six services are defined in `docker-compose.yml`. The four application services (web, two workers, beat) share the same custom Dockerfile. Data stores use official images.

Service Configuration

web — FastAPI application server - Ports: 8000:8000 - Depends on: db (healthy), redis (started)
- Volumes: GCS credentials (read-only)

celery_worker_default — Default queue worker - Command: `celery -A tasks.celery_app worker --concurrency=10 --queues=celery` - Handles: API calls, audio downloads, timeout checks, cleanup - Depends on: db (healthy), redis (started)

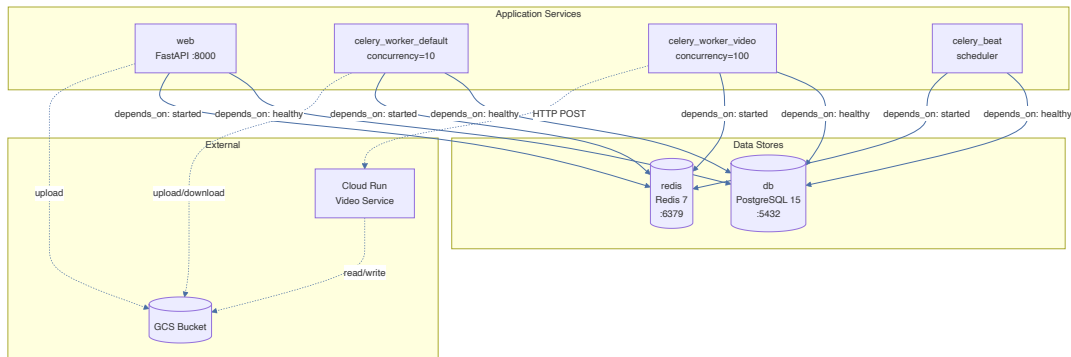
celery_worker_video — Video queue worker - Command: `celery -A tasks.celery_app worker --concurrency=100 --queues=video` - Handles: HTTP dispatch to Cloud Run only - Additional env: `CLOUD_RUN_VIDEO_URL`, `VIDEO_SERVICE_API_KEY` - Depends on: db (healthy), redis (started)

celery_beat — Periodic task scheduler - Command: `celery -A tasks.celery_app beat` - Schedules: `check_timeouts` (every 30 min), `cleanup_old_files` (daily 3 AM) - Depends on: db (healthy), redis (started)

db — PostgreSQL 15 - Health check: `pg_isready -U pah -d pah` (interval 5s, 5 retries) - Persistent volume: `postgres_data`

redis — Redis 7 Alpine - Persistent volume: `redis_data`

Dependency Graph



Docker Compose service dependency graph

All application services receive the same base environment variables (`DATABASE_URL`, `REDIS_URL`, `CELERY_BROKER_URL`, `SONAUTO_API_KEY`, `WEBHOOK_BASE_URL`) plus GCS credentials mounted as a read-only volume. The video worker additionally receives `CLOUD_RUN_VIDEO_URL` and `VIDEO_SERVICE_API_KEY`.

Monitoring & Observability

Health Endpoint

GET /api/health returns a JSON payload with the status of Redis, PostgreSQL, and Celery workers. Returns 200 when all checks pass, 503 otherwise. Designed for load balancer health checks.

Admin Queue Status

GET /api/admin/queue-status returns: - Count of pending submissions - Count of processing submissions - Sonauto API credit balance (cached in Redis for 15 minutes; fetched live on cache miss)

Admin Data Endpoint

GET /api/admin/data implements DataTables server-side processing (SSP) with search, sorting, and pagination. Supports filtering across owner_name, pet_name, and session_id.

Structured Logging

All components use Python's logging module with a consistent format:

```
%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

Key events logged at INFO level: - Submission creation and dispatch - Sonauto API calls (send, fetch, response) - Webhook receipt and chain triggering - GCS uploads/downloads with blob paths and byte counts - Video generation start/completion with timing - Timeout and failure events (WARNING level)

Database Timestamp Audit Trail

Each submission records timestamps at critical pipeline stages: - created_at — when the user submitted the form - sent_to_LLM — when the prompt was sent to Sonauto - streaming_ready_at — when early audio playback became available - received_from_LLM — when the webhook arrived - video_creation_start / video_creation_end — video render duration

This allows post-hoc analysis of pipeline latency at each stage.

Storage Architecture (GCS)

Bucket Structure

All files reside in a single GCS bucket (vday2026) organized by type:

```
vday2026/
├── uploads/          # Pet photos (JPEG, ~50–200 KB each)
│   └── {session_id}.jpg
├── audio/            # Generated songs (MP3, ~3–5 MB each)
│   └── {session_id}.mp3
├── video/            # Generated videos (MP4, ~5–15 MB each)
│   └── {session_id}.mp4
└── images/           # Composite record images (PNG)
    └── {session_id}.png
```

Naming Convention

All files use {session_id} as the filename — a CUID2 identifier that is unique, URL-safe, and collision-resistant. This eliminates the need for name deduplication logic.

URL Patterns

- **Public URLs** for streaming/display: `https://storage.googleapis.com/vday2026/{folder}/{session_id}`
- **Signed download URLs** for the download button: Time-limited (60 min), include `Content-Disposition: attachment` header with a human-readable filename (`{pet-name}-love-song.mp4`)

Client Architecture

The GCS client uses a lazy-loaded singleton pattern:

```
_client: Client | None = None
_bucket: Bucket | None = None

def get_client() -> Client:
    global _client
    if _client is None:
        _client = storage.Client.from_service_account_json(...)
    return _client
```

This avoids repeated authentication handshakes while still supporting both service account credentials (Docker/production) and application default credentials (Cloud Run).

Reusability Guide

This architecture can be adapted for future campaigns with different external APIs and post-processing pipelines. Key adaptation points:

- **Swap the external API:** Replace `send_to_sonauto` and the webhook handler with calls to any async API that supports webhook callbacks. The chain pattern (fetch details -> download artifact -> post-process) is generic.
- **Swap post-processing:** Replace the video generation step with any CPU-intensive output creation (image collage, PDF generation, etc.). The Cloud Run offloading pattern works for any workload that benefits from horizontal scaling.
- **Reuse queue topology:** The two-queue model (default + heavy-processing) is reusable whenever you have a mix of fast tasks (API calls, DB ops) and slow I/O-bound dispatch tasks. Adjust concurrency numbers based on the external service's capacity.
- **Reuse timeout strategy:** The two-layer timeout (event-based countdown + periodic sweep) applies to any webhook-dependent workflow. Adjust `WEBHOOK_TIMEOUT_MINUTES` and the Beat sweep interval to match the external API's expected response time.
- **Adapt rate limiting:** The cookie-based rate limiting is simple but effective for campaign microsites where user accounts don't exist. For authenticated flows, replace `cookie_id` with user IDs.
- **Adapt content safety:** The pluggable image safety backend (Gemini vs NudeNet) pattern can be extended with additional methods. The profanity filter's custom banned-words list can be swapped per campaign.
- **Config-driven via env vars:** All external URLs, API keys, timeouts, concurrency limits, and retention periods are configurable via environment variables. No code changes needed for infrastructure differences between campaigns.
- **GCS as the data plane:** Using object storage as the shared data plane between services (main backend <-> Cloud Run) avoids the complexity of direct file transfers, shared volumes, or service-to-service streaming. Any service that can read/write to the bucket can participate in the pipeline.

Appendix: Configuration Reference

All settings are defined in `backend/app/config.py` using Pydantic Settings. Values are loaded from environment variables, falling back to defaults.

Variable	Default	Description
DATABASE_URL	postgresql://pah:pah_pah@localhost:5432/pah	PostgreSQL connection URL
REDIS_URL	redis://localhost:6379/0	Redis connection URL
CELERY_BROKER_URL	redis://localhost:6379/0	Celery message broker URL
CELERY_RESULT_BACKEND	redis://localhost:6379/0	Celery result storage URL
SONAUTO_API_URL	https://api.sonauto.ai/v1	Sonauto API base URL
SONAUTO_API_KEY	(empty)	Sonauto API authentication key
WEBHOOK_BASE_URL	http://localhost:8000	Base URL for webhook callbacks
MAX_CONCURRENT_REQUESTS	10	Maximum concurrent API requests
MAX_RETRIES	3	Application-level retry limit
FORM_SUBMIT_RETRY	10	Maximum submissions per cookie_id
MIN_AVAILABLE_CREDITS	5000	Minimum Sonauto credits before alert
WEBHOOK_TIMEOUT_MINUTES	10	Minutes before a submission is considered timed out
API_REQUEST_TIMEOUT_SECONDS	10	HTTP timeout for outbound API calls
IMAGE_SAFETY_METHOD	gemini	Image classification backend (gemini or nudenet)
GEMINI_API_KEY	(empty)	Google Gemini API key for image safety
CLOUD_RUN_VIDEO_URL	(empty)	Cloud Run video service URL (empty = local FFmpeg)
VIDEO_SERVICE_API_KEY	(empty)	Shared secret for Cloud Run authentication
CLOUD_RUN_VIDEO_TIMEOUT	300	Cloud Run request timeout in seconds
GCS_PROJECT_ID	holiday-project-india	Google Cloud project ID
GCS_BUCKET_NAME	vday2026	GCS bucket name
GCS_CREDENTIALS_PATH	google_cloud_storage.json	Path to GCS service account JSON
GCS_UPLOADS_FOLDER	uploads	GCS folder for pet photos
GCS_AUDIO_FOLDER	audio	GCS folder for generated audio
GCS_VIDEO_FOLDER	video	GCS folder for generated video
GCS_IMAGES_FOLDER	images	GCS folder for composite images
STORAGE_BASE	../storage	Local storage path (dev fallback)
FILE_RETENTION_DAYS	30	Days before files are cleaned up
CORS_ORIGINS	localhost:8000, :8080	Allowed CORS origins