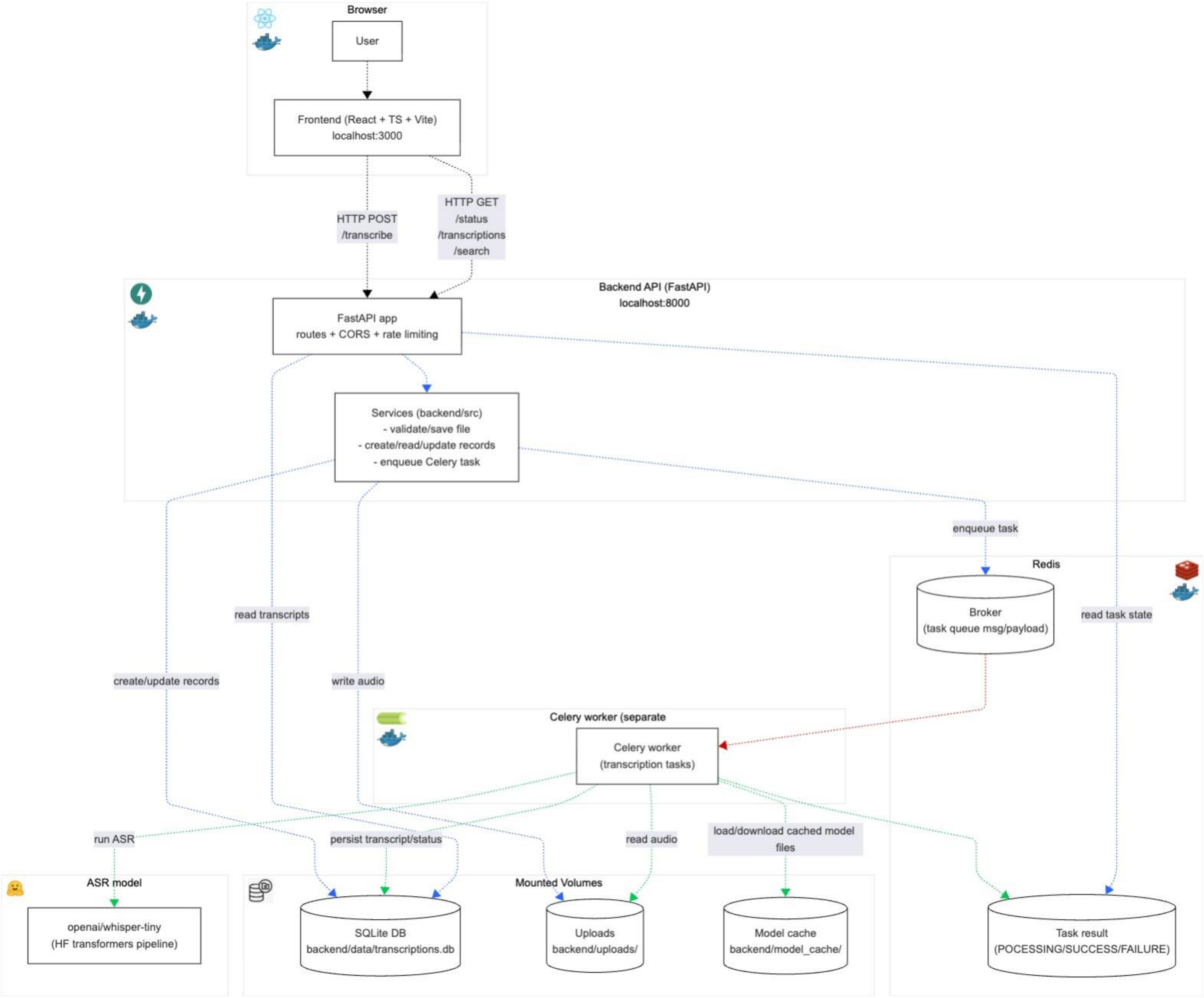
Asher Chew

GitHub Repo: https://github.com/Asherchewzy/asr_transcription/

Architecture Diagram



Architecture Diagram Explained

Frontend (docker container):

- User accesses the React (Vite + TS) frontend on localhost:3000
- Frontend uploads audio with POST /transcribe to the FastAPI backend on localhost:8000
- Frontend polls GET /status, /transcriptions, /search for progress and results.

Backend (docker container):

- FastAPI handles routes, CORS, and rate limiting;
- services validate/save files, create/update records, and enqueue Celery tasks.

Redis (docker container):

- Redis acts as the task broker and stores task results (processing/success/failure).

Celery (docker container):

- Celery worker reads uploaded audio from the shared volume.
- The worker loads the Whisper-tiny model (cached on disk) and runs transcription via Huggingface pipeline.
- Transcription results and status are saved to the SQLite database.

Mounted Volumes:

- Shared volumes persist uploads, the database, and model cache across containers.

Assumptions, Considerations and Decisions

1. The system is designed to **run as a small, single-instance service** that performs audio transcription using a tiny Whisper model **on CPU**. **Files are generally small, and users do not upload large quantities at once.**

   - The service uses whisper-tiny with CPU-based inference by default, with optional GPU support available through Dockerfile.gpu and docker-compose-gpu.yaml.

   - CPU is sufficient for the expected workload, keeping the default deployment simple, while GPU support remains available for environments with NVIDIA hardware.

   - Although files are expected to be small, strict limits are enforced to protect memory and performance on the small instance. The frontend limits uploads to **10 MB**, while the backend enforces a **15 MB** limit as a safety net. This results in an effective 10 MB maximum. Oversized files are blocked in the UI before reaching the server. Even though Whisper can process audio in chunks, size limits prevent excessive memory usage and long processing times. These limits are configurable via environment variables.

- Because file batches are small, only limited background parallelism is required. **Celery** is configured with **two worker processes**, allowing up to two transcription jobs to run simultaneously while balancing CPU and memory usage.

- To protect the API from excessive traffic, **SlowAPI** enforces a limit of **15 successful requests per rolling 60-second window**, preventing request spikes. This is also configurable via environment variables.

2. The system supports **single or batched audio file uploads**, with the assumption that users will typically upload **small batches (around 10–15 files)** rather than hundreds.

   - Since transcription is slow, it is handled asynchronously. **Celery with Redis** is used as a task queue so that transcription jobs do not block HTTP requests.

   - As a result, POST /transcribe returns a task ID instead of the final result. A separate endpoint, GET /status/{task_id}, provides progress updates such as *"Transcribing audio"* or *"Saving results"*.

3. No cloud services are used. **Persistent volume mounts** are used for uploaded files and the database, providing durable storage while keeping the system simple.

4. The instructions mentioned "Files uploaded should be modified to be unique, if the file name already exists in the backend." Thus, the assumption is made that **file can be duplicated but file names must be unique.**

   - The app generates unique filenames using the cleaned original stem, a timestamp, and a random 8-character hex token. This guarantees uniqueness, avoids collisions, and allows filenames to act as primary keys in the database while remaining searchable.

5. Only **MP3** files are accepted. Although Whisper supports multiple formats, restricting to MP3 simplifies validation and reduces complexity.

6. Because users are assumed to be generally responsible, only **lightweight security controls** are applied. These include:

   - Rejecting non-MP3 files, verifying file types using MIME detection (python-magic), validating file size via chunked reads, sanitizing filenames with Google RE2 regular expressions, sanitizing search inputs with Google RE2 regular expressions

   - These measures prevent common issues such as memory exhaustion, path traversal, XSS, and SQL injection without adding unnecessary complexity. See **backend/notebook/sanitizer.ipynb** for more elaboration.

Backend testing

1. test_file_upload

What's being tested

- Concurrent uploads succeed: 10 simultaneous /api/v1/transcribe uploads return 200 and save files.
- Concurrency safety: Unique filenames and unique DB transcription IDs under simultaneous uploads, plus thread-safe generate_unique_filename.
- Error isolation: Mixed valid/invalid files show invalids fail without breaking valid uploads.
- No partial writes: Failed uploads don't leave stray files on disk.
- Batch upload in one request: Multiple files in a single POST return all tasks and save all files.
- Under small load: Burst of requests yields valid 200/429 responses (no crashes/timeouts).
- Validation checks: Oversized files (413), invalid extension (400), MIME spoofing (400), and filename sanitization (path traversal, null bytes, special chars).
- Rate limiting: Ensures limiter actually returns 429s when enabled.

Why it's important

- Protects against race conditions and data corruption in file naming, disk I/O, and DB writes.
- Prevents security issues (path traversal, null bytes, spoofed MIME/ext).
- Ensures robustness under load and correct rate limiting behavior.
- Data hygiene by avoiding partial/garbage files on failures.
- Validates API contract for multi-file requests and task ID generation.

2. test_celery_task_end_to_end_with_real_db

What's being tested

- End-to-end task flow with in-memory SQLite database like a real db: transcribe_audio_task updates DB records through the full lifecycle.
- Happy path: status → completed, text saved, correct result dict returned.
- State updates: task reports PROCESSING via update_state.
- Failure paths: exceptions (generic and FileNotFoundError) set status → failed and capture error messages.
- Edge case: empty transcription still marks as completed.
- Data consistency: both transcribed_text and status updated correctly.
- Isolation: sequential tasks don't cross-contaminate records.

Why it's important

- Ensures the core async workflow (Celery + DB) works end-to-end.
- Catches transaction/session lifecycle issues that in background task execution.
- Guarantees reliable status reporting and error propagation, which the API and UI depend on.
- Validates data integrity across multiple tasks, preventing corrupt or mixed results.

3. test_health_check_dependency_failures

What's being tested

- /api/v1/health returns 200 + healthy when Whisper, DB, Redis/broker, and Celery workers are all OK (and issues is empty).
- Each individual dependency failure returns 503 + degraded, with the correct boolean flag and specific issues code (whisper_model_unloaded, database_unhealthy, redis_unhealthy, c elery_workers_inactive).
- Multiple simultaneous failures still return a valid 503 response with all relevant issue codes.
- Response includes a fresh timestamp, device_info, and all required schema fields.

Why it's important

- Guarantees monitoring can rely on HTTP 503 for unhealthy states. Guarantees HTTP 200 really meant that all services are ready and working.
- Provides actionable diagnostics via issues and flags instead of vague "down" signals.
- Ensures the health endpoint remains stable and contract-complete, even when dependencies fail.

Frontend testing

1. test_complete_user_journey

What's being tested

- End-to-end UI flow: app load → upload → polling → completion → list refresh → search → clear completed.
- Failure path: upload completes with a failed transcription and the UI shows "failed".
- Initial list rendering: existing completed/failed transcriptions display with counts and status badges.
- Post-upload refresh: list is reloaded after a task completes.
- Search behavior: search button and Enter key both trigger search; clear resets to full list.
- Loading and error states: shows loading during fetch; handles API errors without crashing.
- Multiple sequential uploads: back-to-back uploads trigger separate API calls.

Why it's important

- Validates the core product promises: users can upload, get a transcription, and find it.
- Ensures component integration (upload, polling, list, search) works together, not just in isolation.
- Confirms resilience and UX under failures, latency, and repeated actions.
- Guards against regressions in state updates and list refresh logic.

2. test_multi_file_orchestration_user_flow

What's being tested

- Multi-file upload orchestration with 5 concurrent files and mixed outcomes (completed, failed, timed-out), ensuring each polling loop is independent.
- Per-file status rendering: completed vs failed vs processing display correctly and independently.
- Clearing completed/failed items doesn't affect in-progress uploads.
- Client-side validation failures block uploads and surface error messages.

- Upload list preserves file order.
- Upload API failure is handled gracefully without calling onUploadComplete.
- Rapid sequential uploads keep task polling isolated and complete per file.
- Empty file list is a no-op (no upload triggered).

Why it’s important

- Validates concurrency correctness in the core multi-file UX (no cross-talk between tasks).
- Ensures robust state management under mixed outcomes and ongoing polling.
- Protects user experience by keeping list order, clear actions, and error handling predictable.
- Prevents bad uploads (validation failures/empty input) from hitting the backend.

3. test_polling_memory_leak_prevention

What’s being tested

- Polling stops when the component unmounts during an in-progress upload (no continued getTaskStatus calls).
- Timers are cleared/neutralized after unmount to prevent leaks and “setState on unmounted” warnings.
- Polling ends correctly on completed and failed statuses; completion triggers onUploadComplete, failure does not.
- Long-running polls time out after ~60 attempts (≈5 minutes) and show a “Timeout” error.
- API errors during polling stop further polling and are handled without crashes.
- Multiple concurrent uploads don’t interfere with each other (one can finish while the other keeps polling).

Why it’s important

- Checks for memory leaks and runaway timers in a common React pitfall.
- Ensures correct lifecycle cleanup when users navigate away mid-upload.
- Protects app stability and performance under long or failed transcriptions.
- Confirms independent task handling for concurrent uploads.