# Whirly Backend Repo

This document contains a ready-to-run backend repository for **Whirly**: FastAPI service + Postgres + Redis + validator worker + retrain job, Dockerfiles, and docker-compose for local development.

**Note:** This is a full repo scaffold. Copy the file contents into a directory and run docker compose up --build as described in the README below.

## Repo layout

whirly-backend/  
├─ backend/  
│ ├─ app/  
│ │ ├─ \_\_init\_\_.py  
│ │ ├─ main.py  
│ │ ├─ api.py  
│ │ ├─ db.py  
│ │ ├─ models.sql  
│ │ ├─ workers/  
│ │ │ ├─ validator.py  
│ │ │ └─ retrain\_job.py  
│ │ └─ utils.py  
│ ├─ Dockerfile  
│ └─ requirements.txt  
├─ docker/  
│ └─ .env.example  
├─ docker-compose.yml  
├─ Makefile  
└─ README.md

## Files

### backend/app/\_\_init\_\_.py

# empty - package marker

### backend/app/db.py

# db.py - async DB + Redis connection helpers  
import os  
import asyncpg  
import aioredis  
from typing import Optional  
  
DATABASE\_URL = os.getenv("DATABASE\_URL", "postgresql://postgres:postgres@db:5432/whirly")  
REDIS\_URL = os.getenv("REDIS\_URL", "redis://redis:6379/0")  
  
\_db\_pool: Optional[asyncpg.pool.Pool] = None  
\_redis = None  
  
async def get\_db\_pool():  
 global \_db\_pool  
 if \_db\_pool is None:  
 \_db\_pool = await asyncpg.create\_pool(DATABASE\_URL, min\_size=1, max\_size=10)  
 return \_db\_pool  
  
async def get\_redis():  
 global \_redis  
 if \_redis is None:  
 \_redis = await aioredis.from\_url(REDIS\_URL)  
 return \_redis

### backend/app/models.sql

-- SQL schema for initial tables  
CREATE TABLE IF NOT EXISTS inferences (  
 image\_id TEXT PRIMARY KEY,  
 inference\_json JSONB,  
 svg TEXT,  
 raster\_path TEXT,  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT now()  
);  
  
CREATE TABLE IF NOT EXISTS corrections (  
 id BIGSERIAL PRIMARY KEY,  
 image\_id TEXT NOT NULL,  
 user\_id TEXT,  
 timestamp TIMESTAMP WITH TIME ZONE DEFAULT now(),  
 original\_inference JSONB NOT NULL,  
 correction JSONB NOT NULL,  
 status TEXT NOT NULL DEFAULT 'raw',  
 validation\_info JSONB,  
 dataset\_shard TEXT,  
 processing\_notes TEXT  
);  
  
CREATE TABLE IF NOT EXISTS curated\_examples (  
 id BIGSERIAL PRIMARY KEY,  
 image\_id TEXT NOT NULL,  
 svg TEXT,  
 raster\_path TEXT,  
 coco\_annotation JSONB,  
 netlist\_json JSONB,  
 created\_from\_correction\_id BIGINT,  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT now()  
);

### backend/app/utils.py

# utils.py - small helpers  
import json  
from typing import Any  
  
class JSONEncoder(json.JSONEncoder):  
 def default(self, obj: Any):  
 try:  
 return super().default(obj)  
 except Exception:  
 return str(obj)  
  
def safe\_json\_dumps(obj):  
 return JSONEncoder().encode(obj)

### backend/app/api.py

# api.py - core FastAPI app: inference lookup and corrections endpoint  
from fastapi import FastAPI, HTTPException  
from pydantic import BaseModel  
from typing import List, Any, Dict  
from .db import get\_db\_pool, get\_redis  
import json  
  
app = FastAPI(title="Whirly Backend")  
  
class ElementCorrection(BaseModel):  
 id: str  
 action: str  
 new\_label: str = None  
 terminals: List[List[float]] = None  
 new\_elements: List[Dict[str, Any]] = None  
  
class CorrectionPayload(BaseModel):  
 image\_id: str  
 user\_id: str = None  
 elements: List[ElementCorrection]  
 annotation\_meta: Dict[str, Any] = {}  
  
@app.on\_event("startup")  
async def startup():  
 app.state.db = await get\_db\_pool()  
 app.state.redis = await get\_redis()  
  
@app.get("/api/inference/{image\_id}")  
async def get\_inference(image\_id: str):  
 async with app.state.db.acquire() as conn:  
 row = await conn.fetchrow("SELECT inference\_json, svg, raster\_path FROM inferences WHERE image\_id=$1", image\_id)  
 if not row:  
 raise HTTPException(status\_code=404, detail="inference not found")  
 return {"image\_id": image\_id, "svg": row['svg'], "components": row['inference\_json'].get('components', [])}  
  
@app.post("/api/corrections", status\_code=201)  
async def post\_correction(payload: CorrectionPayload):  
 async with app.state.db.acquire() as conn:  
 row = await conn.fetchrow("SELECT inference\_json, svg, raster\_path FROM inferences WHERE image\_id=$1", payload.image\_id)  
 if not row:  
 raise HTTPException(status\_code=404, detail="image not found")  
 q = """  
 INSERT INTO corrections (image\_id, user\_id, original\_inference, correction, status)  
 VALUES ($1,$2,$3,$4,'raw') RETURNING id  
 """  
 rec\_id = await conn.fetchval(q, payload.image\_id, payload.user\_id, row['inference\_json'], json.dumps(payload.dict()))  
 # enqueue  
 await app.state.redis.lpush('corrections\_queue', rec\_id)  
 return {"correction\_id": rec\_id}

### backend/app/main.py

# main.py - run using uvicorn  
import uvicorn  
from .api import app  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 uvicorn.run(app, host="0.0.0.0", port=8000, log\_level="info")

### backend/app/workers/validator.py

# validator.py - async worker to pop corrections, validate, and store curated examples  
import asyncio  
import asyncpg  
import aioredis  
import json  
from shapely.geometry import Point, Polygon  
from ..db import DATABASE\_URL, REDIS\_URL  
  
async def conversion\_function(original, correction):  
 # Stub: convert correction + original -> COCO annotation + netlist  
 # In prod this must be deterministic and robust.  
 coco = {  
 "images": [{"id": 1, "file\_name": original.get('raster\_path', ''), "height": original.get('image\_height', 1024), "width": original.get('image\_width', 1024)}],  
 "annotations": []  
 }  
 return coco  
  
async def validate\_and\_store(conn, correction\_row):  
 correction = correction\_row['correction']  
 original = correction\_row['original\_inference']  
 results = {"valid": True, "issues": []}  
 # simple example check  
 for el in correction['elements']:  
 if el.get('terminals'):  
 for t in el['terminals']:  
 x,y = t  
 if x < 0 or y < 0:  
 results['valid'] = False  
 results['issues'].append(f"terminal out of bounds {t}")  
 if results['valid']:  
 coco = await conversion\_function(original, correction)  
 await conn.execute("INSERT INTO curated\_examples (image\_id, svg, raster\_path, coco\_annotation, netlist\_json, created\_from\_correction\_id) VALUES ($1,$2,$3,$4,$5,$6)", correction['image\_id'], original.get('svg'), original.get('raster\_path'), json.dumps(coco), json.dumps(original.get('netlist')), correction\_row['id'])  
 await conn.execute("UPDATE corrections SET status='validated', validation\_info=$1 WHERE id=$2", json.dumps(results), correction\_row['id'])  
 else:  
 await conn.execute("UPDATE corrections SET status='rejected', validation\_info=$1 WHERE id=$2", json.dumps(results), correction\_row['id'])  
  
async def worker\_loop():  
 redis = await aioredis.from\_url(REDIS\_URL)  
 pool = await asyncpg.create\_pool(DATABASE\_URL)  
 while True:  
 item = await redis.brpop('corrections\_queue', timeout=5)  
 if item:  
 \_, cid = item  
 async with pool.acquire() as conn:  
 row = await conn.fetchrow('SELECT \* FROM corrections WHERE id=$1', int(cid))  
 if row:  
 await validate\_and\_store(conn, row)  
 else:  
 await asyncio.sleep(1)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 asyncio.run(worker\_loop())

### backend/app/workers/retrain\_job.py

# retrain\_job.py - simplified retrain orchestrator (calls training container/script)  
import subprocess  
import os  
import json  
  
# Example: this script exports curated examples and triggers training in a trainer container  
DATASET\_DIR = os.getenv('DATASET\_DIR', '/data/datasets/whirly')  
TRAIN\_SCRIPT = os.getenv('TRAIN\_SCRIPT', '/workspace/train\_mask2former.py')  
OUTPUT\_DIR = os.getenv('MODEL\_OUT', '/models')  
  
def export\_curated\_examples(limit=500):  
 # In production we would query Postgres, write COCO dataset to disk  
 print('Exporting curated examples - stub')  
  
def run\_training():  
 cmd = ["python", TRAIN\_SCRIPT, "--output", OUTPUT\_DIR]  
 print('Running training:', ' '.join(cmd))  
 subprocess.run(cmd, check=True)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 export\_curated\_examples()  
 try:  
 run\_training()  
 except subprocess.CalledProcessError as e:  
 print('Training failed', e)

### backend/requirements.txt

fastapi==0.95.0  
uvicorn[standard]==0.22.0  
asyncpg==0.29.0  
aioredis==2.0.1  
sqlalchemy==1.4.52  
pydantic==1.10.7  
python-dotenv==1.0.0  
shapely==2.0.1  
pycocotools==2.0.6  
cairosvg==2.6.0  
svgwrite==1.4.1  
Pillow==9.5.0

### backend/Dockerfile

FROM python:3.11-slim  
WORKDIR /app  
COPY requirements.txt .  
RUN apt-get update && apt-get install -y build-essential libcairo2 libpango1.0-0 libgdk-pixbuf2.0-0 libffi-dev && rm -rf /var/lib/apt/lists/\*  
RUN pip install --no-cache-dir -r requirements.txt  
COPY app/ ./app  
ENV PYTHONPATH="/app"  
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

### docker/.env.example

POSTGRES\_USER=postgres  
POSTGRES\_PASSWORD=postgres  
POSTGRES\_DB=whirly  
DATABASE\_URL=postgresql://postgres:postgres@db:5432/whirly  
REDIS\_URL=redis://redis:6379/0

### docker-compose.yml

version: '3.8'  
services:  
 db:  
 image: postgres:15  
 environment:  
 POSTGRES\_USER: postgres  
 POSTGRES\_PASSWORD: postgres  
 POSTGRES\_DB: whirly  
 volumes:  
 - db\_data:/var/lib/postgresql/data  
 ports:  
 - "5432:5432"  
  
 redis:  
 image: redis:7  
 ports:  
 - "6379:6379"  
  
 backend:  
 build: ./backend  
 env\_file:  
 - ./docker/.env.example  
 depends\_on:  
 - db  
 - redis  
 ports:  
 - "8000:8000"  
 volumes:  
 - ./backend/app:/app/app  
  
 validator:  
 build: ./backend  
 command: ["python","-u","app/workers/validator.py"]  
 env\_file:  
 - ./docker/.env.example  
 depends\_on:  
 - backend  
 - db  
 - redis  
 volumes:  
 - ./backend/app:/app/app  
  
 retrain:  
 build: ./backend  
 command: ["python","-u","app/workers/retrain\_job.py"]  
 env\_file:  
 - ./docker/.env.example  
 depends\_on:  
 - validator  
 volumes:  
 - ./backend/app:/app/app  
  
volumes:  
 db\_data:

### Makefile

.PHONY: up down logs initdb  
  
up:  
 docker compose up --build  
  
down:  
 docker compose down  
  
logs:  
 docker compose logs -f  
  
initdb:  
 # Run SQL schema  
 docker compose run --rm backend bash -c "psql $${DATABASE\_URL} -f app/models.sql"

### README.md

# Whirly Backend