# Whirly — Traceable Circuit Diagrams: Full 4-piece Bundle

This document contains four ready-to-drop-in deliverables for **whirly**:

1. **MVP FastAPI + OpenCV backend** (accepts uploads, returns SVG + adjacency JSON, trace endpoint)
2. **React + SVG frontend** (consumes JSON, renders SVG, click-to-trace + simple correction UI)
3. **Training notebook** (Jupyter-style guide to synthesize circuit images and train a symbol detector)
4. **PR-style patch** (diff) you can apply to your whirly repo to add the upload endpoint and wire-trace flow.

**Notes:**

* These are *engineer-ready* starter files. The backend uses classical image-processing as an MVP (skeletonize wires, contour-based component detection) so you can iterate quickly. Replace the placeholder detection with YOLO/Detectron2 or Mask R-CNN later (the training notebook shows how).
* Save these files into your project directory. I recommend creating a new whirly-trace-mvp/ subfolder for the backend and whirly-frontend-trace/ for the front-end when testing locally.

# 1) MVP FastAPI + OpenCV backend

**Files and paths**

* backend/app.py — FastAPI app
* backend/cv\_utils.py — image processing helpers
* backend/requirements.txt

## backend/requirements.txt

fastapi  
uvicorn[standard]  
python-multipart  
pillow  
opencv-python  
numpy  
scikit-image  
networkx

## backend/cv\_utils.py

# backend/cv\_utils.py  
import cv2  
import numpy as np  
from skimage.morphology import skeletonize  
import networkx as nx  
from shapely.geometry import LineString, Point  
import json  
  
def preprocess\_image\_bytes(image\_bytes):  
 # Load bytes into OpenCV  
 nparr = np.frombuffer(image\_bytes, np.uint8)  
 img = cv2.imdecode(nparr, cv2.IMREAD\_COLOR)  
 gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
 # denoise  
 gray = cv2.bilateralFilter(gray, 9, 75, 75)  
 # adaptive threshold  
 th = cv2.adaptiveThreshold(gray,255,cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,cv2.THRESH\_BINARY\_INV,15,4)  
 return img, gray, th  
  
def extract\_wire\_skeleton(bin\_img):  
 # bin\_img: binary image where wires are white (255)  
 bw = (bin\_img > 0).astype(np.uint8)  
 # skeletonize expects boolean  
 sk = skeletonize(bw==1).astype(np.uint8)\*255  
 return sk  
  
# Simple contour-based "component" detection (MVP placeholder)  
COMPONENT\_MIN\_AREA = 80  
  
def detect\_components(bin\_img, orig\_img):  
 # find contours in bin image (components often have enclosed shapes)  
 contours, \_ = cv2.findContours(bin\_img.copy(), cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  
 components = []  
 h,w = bin\_img.shape  
 for i,c in enumerate(contours):  
 area = cv2.contourArea(c)  
 if area < COMPONENT\_MIN\_AREA:  
 continue  
 x,y,ww,hh = cv2.boundingRect(c)  
 # crude heuristics to reject thin wire-like contours  
 if ww < 6 or hh < 6:  
 continue  
 comp = {  
 'id': f'c{i}',  
 'bbox':[int(x),int(y),int(ww),int(hh)],  
 'area': float(area),  
 'label':'unknown',  
 }  
 components.append(comp)  
 return components  
  
# Convert skeleton to graph (pixel nodes at junctions/endpoints)  
OFFSETS = [(-1,0),(1,0),(0,-1),(0,1),(-1,-1),(-1,1),(1,-1),(1,1)]  
  
def skeleton\_to\_graph(sk):  
 h,w = sk.shape  
 sk\_bool = (sk>0).astype(np.uint8)  
 G = nx.Graph()  
 node\_map = {}  
 # find junctions/endpoints  
 for y in range(h):  
 for x in range(w):  
 if sk\_bool[y,x]:  
 neighbors = 0  
 for dx,dy in OFFSETS:  
 nxp, nyp = x+dx, y+dy  
 if 0<=nxp<w and 0<=nyp<h and sk\_bool[nyp,nxp]:  
 neighbors += 1  
 if neighbors != 2:  
 nid = len(node\_map)  
 node\_map[(x,y)] = nid  
 G.add\_node(nid, xy=(int(x),int(y)))  
 # walk edges from each node  
 visited\_edges = set()  
 for (sx,sy), nid in list(node\_map.items()):  
 for dx,dy in OFFSETS:  
 nxp, nyp = sx+dx, sy+dy  
 if not (0<=nxp<w and 0<=nyp<h and sk\_bool[nyp,nxp]):  
 continue  
 path = [(sx,sy)]  
 cx,cy = nxp,nyp  
 prev = (sx,sy)  
 while True:  
 path.append((cx,cy))  
 if (cx,cy) in node\_map and (cx,cy) != (sx,sy):  
 nid2 = node\_map[(cx,cy)]  
 edge\_id = tuple(sorted((nid, nid2)))  
 if edge\_id not in visited\_edges:  
 visited\_edges.add(edge\_id)  
 G.add\_edge(nid, nid2, pixels=path)  
 break  
 # step forward  
 moved = False  
 for ddx,ddy in OFFSETS:  
 tx,ty = cx+ddx, cy+ddy  
 if 0<=tx<w and 0<=ty<h and sk\_bool[ty,tx] and (tx,ty) != prev:  
 prev = (cx,cy)  
 cx,cy = tx,ty  
 moved = True  
 break  
 if not moved:  
 break  
 return G  
  
# Snap components to nearest node within tolerance  
def associate\_components\_to\_nodes(components, graph, tolerance=12):  
 # graph nodes have 'xy'  
 for comp in components:  
 x,y,w,h = comp['bbox']  
 cx = x + w/2  
 cy = y + h/2  
 nearest = None  
 ndist = None  
 for n,d in graph.nodes(data=True):  
 nx\_,ny\_ = d['xy']  
 dist = (nx\_-cx)\*\*2 + (ny\_-cy)\*\*2  
 if ndist is None or dist < ndist:  
 ndist = dist  
 nearest = n  
 if ndist is not None and (ndist\*\*0.5) <= tolerance:  
 comp['node'] = nearest  
 else:  
 comp['node'] = None  
 return components  
  
# Export simple SVG and JSON graph  
SVG\_TEMPLATE = '<svg xmlns="http://www.w3.org/2000/svg" width="{w}" height="{h}" viewBox="0 0 {w} {h}">\n{content}\n</svg>'  
  
def graph\_to\_json(graph, components):  
 nodes = []  
 edges = []  
 for n,d in graph.nodes(data=True):  
 nodes.append({'id':int(n),'xy':d['xy']})  
 for u,v,d in graph.edges(data=True):  
 edges.append({'u':int(u),'v':int(v),'pixels':d.get('pixels',[])})  
 return {'nodes':nodes,'edges':edges,'components':components}  
  
  
def graph\_to\_svg(graph, components, width, height):  
 parts = []  
 # draw edges as polyline  
 for i,(u,v,d) in enumerate(graph.edges(data=True)):  
 pts = d.get('pixels',[])  
 if not pts: continue  
 path = ' '.join([f'{x},{y}' for x,y in pts])  
 parts.append(f'<polyline id="edge{i}" points="{path}" fill="none" stroke="#000" stroke-width="2"/>' )  
 # draw component boxes  
 for comp in components:  
 x,y,w,h = comp['bbox']  
 parts.append(f'<rect id="{comp["id"]}" x="{x}" y="{y}" width="{w}" height="{h}" stroke="red" fill="none" stroke-width="2"/>')  
 return SVG\_TEMPLATE.format(w=width,h=height,content='\n'.join(parts))

## backend/app.py

# backend/app.py  
from fastapi import FastAPI, File, UploadFile  
from fastapi.responses import JSONResponse, HTMLResponse  
from fastapi.staticfiles import StaticFiles  
from cv\_utils import preprocess\_image\_bytes, extract\_wire\_skeleton, detect\_components, skeleton\_to\_graph, associate\_components\_to\_nodes, graph\_to\_json, graph\_to\_svg  
import uvicorn  
  
app = FastAPI()  
  
# mount a simple static folder if you put frontend build here  
app.mount('/static', StaticFiles(directory='static'), name='static')  
  
@app.post('/upload')  
async def upload(file: UploadFile = File(...)):  
 contents = await file.read()  
 orig, gray, th = preprocess\_image\_bytes(contents)  
 sk = extract\_wire\_skeleton(th)  
 graph = skeleton\_to\_graph(sk)  
 components = detect\_components(th, orig)  
 components = associate\_components\_to\_nodes(components, graph)  
 width = orig.shape[1]  
 height = orig.shape[0]  
 svg = graph\_to\_svg(graph, components, width, height)  
 json\_out = graph\_to\_json(graph, components)  
 return JSONResponse({'svg': svg, 'graph': json\_out})  
  
@app.post('/trace')  
async def trace(payload: dict):  
 # payload: { 'start\_node': int }  
 # MVP: do a BFS on posted graph and return list of edge ids to highlight  
 graph = payload.get('graph')  
 start = payload.get('start\_node')  
 if graph is None or start is None:  
 return JSONResponse({'error':'graph and start\_node required'}, status\_code=400)  
 # rebuild adjacency quickly  
 adj = {}  
 for e in graph['edges']:  
 u,v = e['u'], e['v']  
 adj.setdefault(u, []).append((v,e))  
 adj.setdefault(v, []).append((u,e))  
 visited = set()  
 q=[start]  
 visited.add(start)  
 edge\_ids=[]  
 while q:  
 cur = q.pop(0)  
 for (nbr,e) in adj.get(cur,[]):  
 edge\_ids.append(e)  
 if nbr not in visited:  
 visited.add(nbr); q.append(nbr)  
 return JSONResponse({'visited\_nodes':list(visited),'edges':edge\_ids})  
  
@app.post('/save-correction')  
async def save\_correction(payload: dict):  
 # store correction locally for now  
 import os, json  
 os.makedirs('corrections', exist\_ok=True)  
 idx = len([n for n in os.listdir('corrections') if n.endswith('.json')])  
 with open(f'corrections/corr\_{idx}.json','w') as f:  
 json.dump(payload, f)  
 return JSONResponse({'ok':True})  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 uvicorn.run('app:app', host='0.0.0.0', port=8000, reload=True)

# 2) React + SVG frontend

**Files**

* frontend/src/App.jsx — main React app
* frontend/package.json — minimal

## frontend/package.json

{  
 "name": "whirly-frontend-trace",  
 "version": "0.1.0",  
 "private": true,  
 "dependencies": {  
 "react": "18.x",  
 "react-dom": "18.x",  
 "react-scripts": "5.x",  
 "axios": "^1.0.0"  
 },  
 "scripts": {  
 "start": "react-scripts start",  
 "build": "react-scripts build"  
 }  
}

## frontend/src/App.jsx

import React, {useState, useRef} from 'react';  
import axios from 'axios';  
  
export default function App(){  
 const [svgText, setSvgText] = useState(null);  
 const [graph, setGraph] = useState(null);  
 const fileRef = useRef();  
  
 async function handleUpload(e){  
 const f = e.target.files[0];  
 const fd = new FormData();  
 fd.append('file', f);  
 const resp = await axios.post('/upload', fd, { headers:{ 'Content-Type':'multipart/form-data' } });  
 setSvgText(resp.data.svg);  
 setGraph(resp.data.graph);  
 }  
  
 function handleClick(evt){  
 // find nearest node by distance (MVP)  
 if(!graph) return;  
 const svg = evt.currentTarget.ownerSVGElement || evt.currentTarget;  
 const pt = svg.createSVGPoint();  
 pt.x = evt.clientX; pt.y = evt.clientY;  
 const ctm = svg.getScreenCTM().inverse();  
 c