Sprint 2 – Code listing & tool and integration document

London (Tools & Integration Manager)
CS3053 – Software Engineering
Adam, Danny, Ewen, London
1 December 2025

1. Purpose of This Document

This document consolidates:

- All setup and installation steps required to run the Transcriber/Translator system
- A complete list of tools used during Sprint 1 & Sprint 2
- Full source code for the application (FastAPI server, CLI, pipeline, scripts)
- Verification steps to ensure dependencies are installed correctly
- Integration notes confirming everything works together
- This satisfies the Sprint 2 requirement: Provide implementation details and demonstrate how tools were used to support development.

2. Tools & Environment Setup

2.1 Homebrew (macOS package manager)

Used to install FFmpeg and Git.
Website: https://brew.sh/

Check version:

```
brew -version
```

2.2 FFmpeg (audio/video processing tool)

Converts input files (.mp3/.mp4) into 16kHz mono WAV for Whisper.
Website: https://ffmpeg.org/download.html

Install:

```
brew install ffmpeg
```

Verify:

```
ffmpeg –version
```

2.3 Python 3 (main programming language)

Used to implement the server, CLI, and pipeline.
Website: https://www.python.org/downloads/

Check version:

```
python3 –version
```

2.4 Pip (Python package manager)

Used to install required libraries.

Check version:

```
pip3 --version
```

2.5 Git (version control)

Used to push updates to GitHub.
Website: https://git-scm.com/downloads

Install:

```
brew install git
```

2.6 Visual Studio Code (primary editor)

Website: https://code.visualstudio.com/

VS Code Extensions Used:

- Python (Microsoft) – syntax, debugging, running
- Pylance – IntelliSense
- REST Client – API testing

- GitHub Pull Requests (optional)

2.7 Virtual Environment Setup

1. Create virtual environment:

```
python3 -m venv .venv
```

2. Activate:

```
source .venv/bin/activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

4. Run server:

```
uvicorn app:app --reload
```

Open API docs:

http://127.0.0.1:8000/docs

Outputs appear in:

Data/outputs/

3. Dependency Check Instructions

To verify all tools are correctly installed:

```
python check_dependencies.py
```

This checks:

- Python libraries (torch, whisper, transformers, fastapi, uvicorn, python-multipart)
- FFmpeg installation
- Torch backend (CPU/GPU)
- Whisper model load

4. System Usage Instructions

4.1 Running the FastAPI Server:

```
uvicorn app:app --reload
```

Visit:

http://127.0.0.1:8000/docs

Upload .mp3 or .mp4.

4.2 Running the CLI:

```
python cli.py input.mp4 --whisper-model small
```

Optional translation:

```
python cli.py input.mp4 --translate --marian-model Helsinki-NLP/opus-
mt-en-fr
```

5. Full source code (Sprint 2)

5.1 app.py

```
from pathlib import Path
import shutil

from fastapi import FastAPI, UploadFile, File, Form
from fastapi.responses import JSONResponse
from fastapi.middleware.cors import CORSMiddleware

from transcribe_pipeline import process_file, UPLOAD_DIR
```

```
app = FastAPI(
    title="Local Transcriber/Translator",
    version="0.1.0",
    description="Upload .mp3/.mp4, get transcript + captions, all
local.",
)

# Optional: This allows for cross-origin if you later add a web
frontend
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


@app.get("/health")
def health():
    return {"status": "ok"}


@app.post("/transcribe")
async def transcribe_endpoint(
    file: UploadFile = File(...),
    whisper_model: str = Form("small"),
    whisper_lang: str | None = Form(None),
    translate: bool = Form(False),
    marian_model: str | None = Form(None),
):
    """
    Upload an audio/video file and run the local pipeline.
    """
    try:
        # 1) Save uploaded file into UPLOAD_DIR
        upload_path = UPLOAD_DIR / file.filename
        with upload_path.open("wb") as buffer:
            shutil.copyfileobj(file.file, buffer)
        # 2) Run pipeline
        paths = process_file(
            upload_path,
            whisper_model=whisper_model,
            whisper_lang=whisper_lang or None,
            translate=translate,
            marian_model=marian_model,
        )

        return paths

    except Exception as e:
        return JSONResponse(status_code=500, content={"error":
str(e)})
```

5.2 check_dependencies.py:

```
""" check_dependencies.py


Quick script to verify that all required libraries and system
dependencies for the Auto Transcriber project are correctly installed.

Run: python check_dependencies.py """

import importlib import subprocess import sys import shutil
```

=== Libraries to check ===

```
REQUIRED_LIBRARIES = [ "torch", "transformers", "whisper", "fastapi",
"uvicorn", "python_multipart", ]

def check_library(lib_name): try: importlib.import_module(lib_name)
print(f"✅ {lib_name} imported successfully") return True except
ImportError: print(f"❌ {lib_name} not found. Try: pip install
{lib_name}") return False
```

```
def check_ffmpeg(): print("\nChecking FFmpeg installation...")
ffmpeg_path = shutil.which("ffmpeg") if ffmpeg_path: try: result =
subprocess.run(["ffmpeg", "-version"], capture_output=True, text=True)
if result.returncode == 0: print(f"✅ FFmpeg found at: {ffmpeg_path}")
print(result.stdout.splitlines()[0]) return True except Exception as
e: print(f"⚠ Error running FFmpeg: {e}") print("❌ FFmpeg not found.
Install it before running the app.") print(" macOS: brew install
ffmpeg") print(" Ubuntu: sudo apt install ffmpeg") print(" Windows:
choco install ffmpeg or add to PATH") return False

def check_torch_backend(): print("\nChecking PyTorch backend...") try:
import torch device = "cuda" if torch.cuda.is_available() else ( "mps"
if getattr(torch.backends, "mps", None) and
torch.backends.mps.is_available() else "cpu") print(f"✅ Torch
working. Device available: {device}") return True except Exception as
e: print(f"❌ Torch failed: {e}") return False

def check_whisper_model_load(): print("\nTesting Whisper model load
(small)...") try: import whisper model = whisper.load_model("small")
print("✅ Whisper model loaded successfully") del model return True
except Exception as e: print(f"⚠ Whisper model failed to load: {e}")
print(" Check internet connection or Whisper install.") return False

def main(): print("=== Auto Transcriber Environment Check ===\n")

ok_libs = all(check_library(lib) for lib in REQUIRED_LIBRARIES)
ok_ffmpeg = check_ffmpeg()
ok_torch = check_torch_backend()
ok_whisper = check_whisper_model_load()

print("\n=== Summary ===")
if all([ok_libs, ok_ffmpeg, ok_torch, ok_whisper]):
    print("🎉 Environment looks good! You're ready to run the
project.")
else:
    print("⚠  One or more checks failed. Review the messages above
and fix issues before running the app.")


if name == "main": main()
```

5.3 cli.py:

```
#!/usr/bin/env python3 import argparse import json from pathlib import
Path

from transcribe_pipeline import process_file

def main(): parser = argparse.ArgumentParser( description="Local audio
transcriber + (optional) translator." )
parser.add_argument("input_file", help="Path to .mp3/.mp4/.wav file")
parser.add_argument( "--whisper-model", default="small", help="Whisper
model: tiny, base, small, medium, large", ) parser.add_argument( "--
whisper-lang", default=None, help="Force language code (e.g. en, fr).
Default: auto-detect.", ) parser.add_argument( "--translate",
action="store_true", help="Translate transcript using MarianMT.", )
parser.add_argument( "--marian-model", default=None, help="Marian
model name, e.g. Helsinki-NLP/opus-mt-en-fr", )

args = parser.parse_args()

paths = process_file(
    Path(args.input_file),
    whisper_model=args.whisper_model,
    whisper_lang=args.whisper_lang,
    translate=args.translate,
    marian_model=args.marian_model,
)

# Print JSON so other tools can consume it if needed
print(json.dumps(paths, indent=2))


if name == "main": main()
```

5.4 CLI_Install_Commands.txt:

```
CLI installation commands:


1. First Upgrade pip to latest version:
```

```
pip install --upgrade pip
```

2. Install dependencies.

```
pip install torch
pip install openai-whisper
pip install transformers
pip install fastapi
pip install "uvicorn[standard]"
pip install python-multipart
```

3. Verify FFmpeg (macOS / Windows)

- macOS:

  ```
  brew install ffmpeg
  ```

- Windows:

  ```
  choco install ffmpeg
  ```

- Check version (Both):

  ```
  ffmpeg -version
  ```

4. Run "check_dependencies.py" script to ensure everything is downloaded correctly.

5.5 sanity.py:

```
import whisper

from transformers import MarianMTModel

print("Whisper + Marian ready to go")
```

5.6 simple_whisper_test.py:

```
from pathlib import Path import whisper

AUDIO_PATH = Path("data/uploads/harvard.wav") OUTPUT_PATH =
Path("data/outputs/harvard_simple.txt")

def main(): if not AUDIO_PATH.exists(): raise
FileNotFoundError(f"Audio not found at {AUDIO_PATH}")

print("Loading Whisper model 'small'...")
model = whisper.load_model("small")

print(f"Transcribing {AUDIO_PATH} ...")
result = model.transcribe(str(AUDIO_PATH))

text = result["text"].strip()
print("\n=== TRANSCRIPT (first 300 chars) ===")
print(text[:300])
print("\n=== END PREVIEW ===")

OUTPUT_PATH.parent.mkdir(parents=True, exist_ok=True)
OUTPUT_PATH.write_text(text, encoding="utf-8")
print(f"\nSaved full transcript to: {OUTPUT_PATH}")


if name == "main": main()
```

5.7 transcribe_pipeline.py:

```
import json import subprocess import uuid from pathlib import Path
from datetime import timedelta from typing import Optional, List, Dict

import whisper import torch from transformers import MarianMTModel,
MarianTokenizer
```

---------- Paths ----------

```
ROOT = Path(file).resolve().parent DATA_DIR = ROOT / "data" UPLOAD_DIR
= DATA_DIR / "uploads" OUTPUT_DIR = DATA_DIR / "outputs"

UPLOAD_DIR.mkdir(parents=True, exist_ok=True)
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
```

---------- FFmpeg: convert to 16kHz mono WAV ----------

```
def run_ffmpeg_to_wav(input_path: Path) -> Path: """ Convert input
audio/video to 16kHz mono WAV using ffmpeg. """ output_name =
f"{input_path.stem}_{uuid.uuid4().hex[:8]}.wav" output_path =
OUTPUT_DIR / output_name

cmd = [
    "ffmpeg",
    "-y",                 # overwrite existing
    "-i", str(input_path),
    "-ac", "1",           # mono
    "-ar", "16000",       # 16 kHz
    str(output_path),
]

subprocess.run(cmd, check=True)
return output_path
```

---------- Helpers: timestamp formatting ----------

```
def seconds_to_srt_ts(t: float) -> str: """Convert seconds to SRT
timestamp: HH:MM:SS,mmm""" td = timedelta(seconds=float(t))
```

```
total_seconds = int(td.total_seconds()) hours = total_seconds // 3600
minutes = (total_seconds % 3600) // 60 seconds = total_seconds % 60
millis = int(td.microseconds / 1000) return
f"{hours:02d}:{minutes:02d}:{seconds:02d},{millis:03d}"

def seconds_to_vtt_ts(t: float) -> str: """Convert seconds to VTT
timestamp: HH:MM:SS.mmm""" td = timedelta(seconds=float(t))
total_seconds = int(td.total_seconds()) hours = total_seconds // 3600
minutes = (total_seconds % 3600) // 60 seconds = total_seconds % 60
millis = int(td.microseconds / 1000) return
f"{hours:02d}:{minutes:02d}:{seconds:02d}.{millis:03d}"
```

---------- Whisper: local transcription ----------

```
def run_whisper( wav_path: Path, model_name: str = "small", language:
Optional[str] = None, ) -> (List[Dict,], str): """ Run Whisper locally
and return (segments, full_text). Each segment: {id, start, end,
text}. """ model = whisper.load_model(model_name) # uses GPU if
available result = model.transcribe(str(wav_path), language=language)

segments = []
for seg in result["segments"]:
    segments.append(
        {
            "id": seg["id"],
            "start": seg["start"],
            "end": seg["end"],
            "text": seg["text"].strip(),
        }
    )

full_text = result["text"].strip()
return segments, full_text
```

---------- MarianMT: local translation ----------

class Translator: def **init**(self, model_name: str): self.model_name =
model_name self.tokenizer =

```
MarianTokenizer.from_pretrained(model_name) self.model =
MarianMTModel.from_pretrained(model_name)

def translate_texts(self, texts: List[str], max_length: int = 512) ->
List[str]:
    batch = self.tokenizer(
        texts,
        return_tensors="pt",
        padding=True,
        truncation=True,
        max_length=max_length,
    )
    with torch.no_grad():
        generated = self.model.generate(**batch,
max_length=max_length)
    return self.tokenizer.batch_decode(generated,
skip_special_tokens=True)


def translate_segments( segments: List[Dict], model_name: str, ) ->
(List[Dict], str): """ Translate segment texts and return
(translated_segments, full_translated_text). """ translator =
Translator(model_name) texts = [s["text"] for s in segments]
translations = translator.translate_texts(texts)

translated_segments = []
for seg, tr in zip(segments, translations):
    new_seg = dict(seg)
    new_seg["translated_text"] = tr.strip()
    translated_segments.append(new_seg)

full_translated = "\n".join(s["translated_text"] for s in
translated_segments)
return translated_segments, full_translated
```

---------- Output formatters ----------

```
def write_transcript_txt(full_text: str, base_name: str) -> Path: path
= OUTPUT_DIR / f"{base_name}.txt" with path.open("w", encoding="utf-
8") as f: f.write(full_text) return path

def write_srt( segments: List[Dict], base_name: str, use_translated:
bool = False, ) -> Path: path = OUTPUT_DIR / f"{base_name}.srt" with
path.open("w", encoding="utf-8") as f: for i, seg in
enumerate(segments, start=1): start_ts =
seconds_to_srt_ts(seg["start"]) end_ts = seconds_to_srt_ts(seg["end"])
text_field = "translated_text" if use_translated and "translated_text"
in seg else "text" text = seg[text_field].strip()

       f.write(f"{i}\n")
        f.write(f"{start_ts} --> {end_ts}\n")
        f.write(f"{text}\n\n")
return path


def write_vtt( segments: List[Dict], base_name: str, use_translated:
bool = False, ) -> Path: path = OUTPUT_DIR / f"{base_name}.vtt" with
path.open("w", encoding="utf-8") as f: f.write("WEBVTT\n\n") for seg
in segments: start_ts = seconds_to_vtt_ts(seg["start"]) end_ts =
seconds_to_vtt_ts(seg["end"]) text_field = "translated_text" if
use_translated and "translated_text" in seg else "text" text =
seg[text_field].strip()

       f.write(f"{start_ts} --> {end_ts}\n")
        f.write(f"{text}\n\n")
return path


def write_segments_json( segments: List[Dict], base_name: str, ) ->
Path: path = OUTPUT_DIR / f"{base_name}_segments.json" with
path.open("w", encoding="utf-8") as f: json.dump(segments, f,
ensure_ascii=False, indent=2) return path
```

```
---------- Main pipeline ----------

def process_file( input_path: Path, whisper_model: str = "small",
whisper_lang: Optional[str] = None, translate: bool = False,
marian_model: Optional[str] = None, ) -> Dict[str, str]: """ Full
pipeline: input -> ffmpeg (wav) -> Whisper -> optional Marian -> txt,
srt, vtt, segments.json. Returns dict of file paths as strings. """
input_path = input_path.resolve() if not input_path.exists(): raise
FileNotFoundError(f"Input file not found: {input_path}")

# 1) ffmpeg -> wav
wav_path = run_ffmpeg_to_wav(input_path)

# 2) whisper
segments, full_text = run_whisper(
  wav_path,
    model_name=whisper_model,
    language=whisper_lang,)

# 3) optional translate
translated_segments = None
full_translated = None
if translate:
    if not marian_model:
        raise ValueError("marian_model must be provided if
translate=True.")
    translated_segments, full_translated =
translate_segments(segments, marian_model)

caption_segments = translated_segments if translated_segments is not
None else segments
base_name = input_path.stem

# 4) write outputs
main_text = full_translated if full_translated is not None else
full_text
txt_path = write_transcript_txt(main_text, base_name)
srt_path = write_srt(caption_segments, base_name,
use_translated=translate)
```

```
vtt_path = write_vtt(caption_segments, base_name,
use_translated=translate)
segjson_path = write_segments_json(
    translated_segments if translated_segments is not None else
segments,
    base_name,
)

return {
    "transcript_txt": str(txt_path),
    "captions_srt": str(srt_path),
    "captions_vtt": str(vtt_path),
    "segments_json": str(segjson_path),
    "wav_intermediate": str(wav_path),
}
```

5.8 README.md:

Transcription-Website

The Auto Transcriber & Translator is a Python-based application that automatically converts audio or video files into timestamped transcripts and captions, with optional translation into other languages. Built with Whisper for speech recognition and MarianMT for translation, it uses FFmpeg to preprocess input media into standardized 16 kHz mono .wav files. The system outputs synchronized .txt, .srt, .vtt, and .json files—each containing the transcribed or translated text with precise timing data. Users can run it via a simple CLI or interact through a lightweight FastAPI web server. The project demonstrates a complete end-to-end NLP pipeline integrating audio processing, deep learning, and practical deployment, designed for accessibility and reproducibility across platforms.