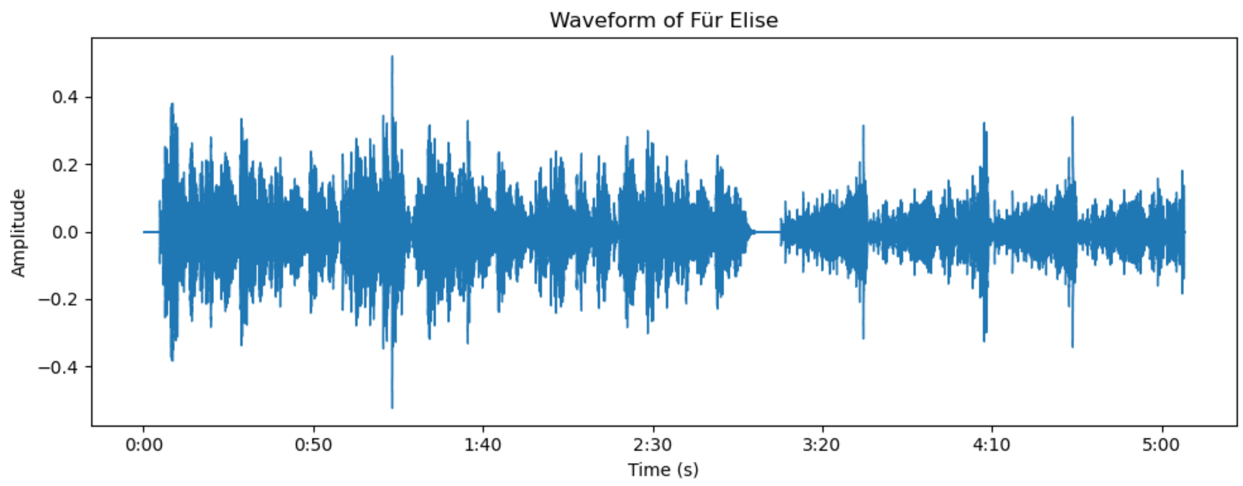
```python
#Visualizing the Waveform

import librosa
import librosa.display
import matplotlib.pyplot as plt

# Load audio
y, sr = librosa.load("fur_elise.mp3", sr=22050)

# Plot waveform
plt.figure(figsize=(10, 4))
librosa.display.waveshow(y, sr=sr)
plt.title("Waveform of Für Elise")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.tight_layout()
plt.show()
```

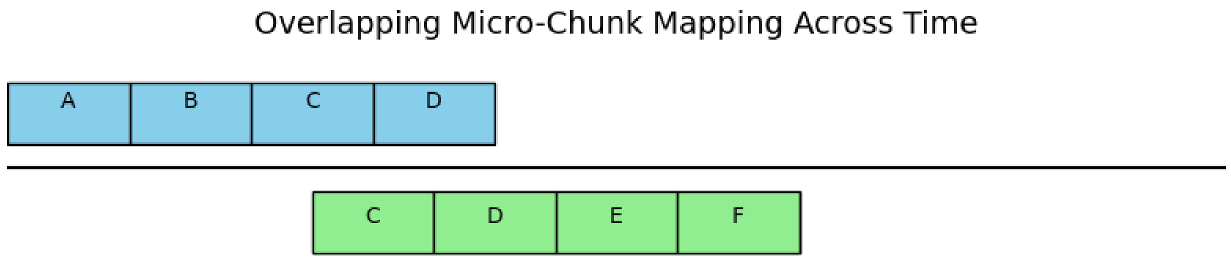


```python
#Pulling metadata of the file

from tinytag import TinyTag

tag = TinyTag.get('fur_elise.mp3')

# Print available metadata
print("Title:", tag.title)
print("Artist:", tag.artist)
print("Duration:", tag.duration, "seconds")
print("Album:", tag.album)
print("Bitrate:", tag.bitrate, "kbps")
```

```
Title: None
Artist: None
Duration: 306.81247740838796 seconds
```

```
Album: None
Bitrate: 128.0 kbps

#Compress the metadata

import zlib

metadata = {
    "ISRC": "TEST99999999",                         # Simulated
    "Title": "Für Elise",                           # Manually added
    "Performer": "Ludwig van Beethoven",            # Known
    "Duration": "05:06",                            # From TinyTag output
(306 sec ≈ 5 min 6 sec)
    "Owner": "Public Domain"                        # Based on the source
}

# Serialize and compress metadata
payload_str = "|".join(f"{k}:{v}" for k, v in metadata.items())
compressed = zlib.compress(payload_str.encode('utf-8'))

print(f"Original size: {len(payload_str.encode('utf-8'))} bytes")
print(f"Compressed size: {len(compressed)} bytes")

Original size: 100 bytes
Compressed size: 101 bytes

#Simulation of 1-second Window Carrying ~4 Chunks

import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Simulated chunk labels
chunk_labels = ['A', 'B', 'C', 'D', 'E', 'F']

fig, ax = plt.subplots(figsize=(10, 2))

# Draw the timeline (0 to 2 seconds)
ax.hlines(1, 0, 2, color='black', linewidth=1.5)
ax.set_ylim(0.9, 1.1)
ax.set_xlim(0, 2)

# First second: Chunks A, B, C, D (0.0–1.0s)
x_start = 0.0
for i, label in enumerate(chunk_labels[:4]):
    rect = patches.Rectangle((x_start + i*0.2, 1.02), 0.2, 0.05,
facecolor='skyblue', edgecolor='black')
    ax.add_patch(rect)
    ax.text(x_start + i*0.2 + 0.1, 1.05, label, ha='center')

# Second second: Chunks C, D, E, F (0.5–1.5s)
x_start = 0.5
```

```
for i, label in enumerate(chunk_labels[2:]):
    rect = patches.Rectangle((x_start + i*0.2, 0.93), 0.2, 0.05,
facecolor='lightgreen', edgecolor='black')
    ax.add_patch(rect)
    ax.text(x_start + i*0.2 + 0.1, 0.955, label, ha='center')

# Labels and title
ax.set_title("Overlapping Micro-Chunk Mapping Across Time",
fontsize=14)
ax.axis('off')
plt.show()
```



```
# Simulates embedding 4 micro-chunks into a 1-second audio slice and
recovers them.

import numpy as np
import zlib
import matplotlib.pyplot as plt
from pydub import AudioSegment

# Simulated Metadata
metadata = {
    "ISRC": "TEST99999999",
    "Title": "Für Elise",
    "Performer": "Beethoven",
    "Duration": "05:06",
    "Owner": "Public Domain"
}

# Compress and Chunk
def compress_and_chunk(metadata_dict, chunk_size=300):
    payload_str = "|".join(f"{k}:{v}" for k, v in
metadata_dict.items())
    compressed = zlib.compress(payload_str.encode('utf-8'))

    chunks = []
    step = chunk_size // 2
    for start in range(0, len(compressed), step):
        chunk = compressed[start:start+chunk_size]
        if len(chunk) < chunk_size:
            chunk = chunk.ljust(chunk_size, b'\0')
        chunks.append(chunk)
    return chunks

# Embed 4 Chunks into Audio Segment
def embed_chunks_in_audio(audio_segment, chunks):
    """
    Simulate embedding: we just store the chunks alongside the audio.
    In real life this would hide chunks in spectrogram or signal.
    """
    embedded = {
        "audio_raw": audio_segment.get_array_of_samples(),
        "frame_rate": audio_segment.frame_rate,
        "chunks": chunks[:4]  # Simulating only 4 for this segment
    }
    return embedded

# Recover from Simulated Audio Segment
def recover_chunks_from_audio(simulated_embedded_audio):
    return simulated_embedded_audio["chunks"]

# Rebuild Metadata from Recovered Chunks
def rebuild_metadata_from_chunks(chunks):
    recovered_bytes = b""
    for c in chunks:
        recovered_bytes += c
    try:
        decompressed = zlib.decompress(recovered_bytes)
        fields = decompressed.decode('utf-8').split('|')
        metadata_recovered = {f.split(':')[0]: f.split(':')[1] for f
in fields if ':' in f}
        return metadata_recovered
    except Exception as e:
        print("Failed to decompress or decode:", e)
        return None

if __name__ == "__main__":
    # Load test audio (ensure it's exactly 1 second for this test)
    audio = AudioSegment.from_file("fur_elise.mp3").set_channels(1)
    one_sec_audio = audio[10000:11000]  # simulate a random 1s window

    chunks = compress_and_chunk(metadata)
    simulated_embedded = embed_chunks_in_audio(one_sec_audio, chunks)

    print("Embedding simulated. Now recovering...")
    recovered_chunks = recover_chunks_from_audio(simulated_embedded)
    reconstructed_metadata =
rebuild_metadata_from_chunks(recovered_chunks)
```

```
    print("\nRecovered Metadata:")
    print(reconstructed_metadata)
```

```
Embedding simulated. Now recovering...

Recovered Metadata:
{'ISRC': 'TEST99999999', 'Title': 'Für Elise', 'Performer':
'Beethoven', 'Duration': '05', 'Owner': 'Public Domain'}
```

```
import zlib

metadata = {
    "ISRC": "TEST99999999",
    "Title": "Für Elise",
    "Performer": "Beethoven",
    "Duration": "05:06",
    "Owner": "Public Domain"
}

def compress_and_chunk(metadata_dict, chunk_size=300):
    payload_str = "|".join(f"{k}:{v}" for k, v in
metadata_dict.items())
    compressed = zlib.compress(payload_str.encode("utf-8"))

    chunks = []
    step = chunk_size // 2
    for start in range(0, len(compressed), step):
        chunk = compressed[start:start + chunk_size]
        if len(chunk) < chunk_size:
            chunk = chunk.ljust(chunk_size, b'\0')
        chunks.append(chunk)
    return chunks

chunks = compress_and_chunk(metadata)
print(f"Total chunks created: {len(chunks)}")
```

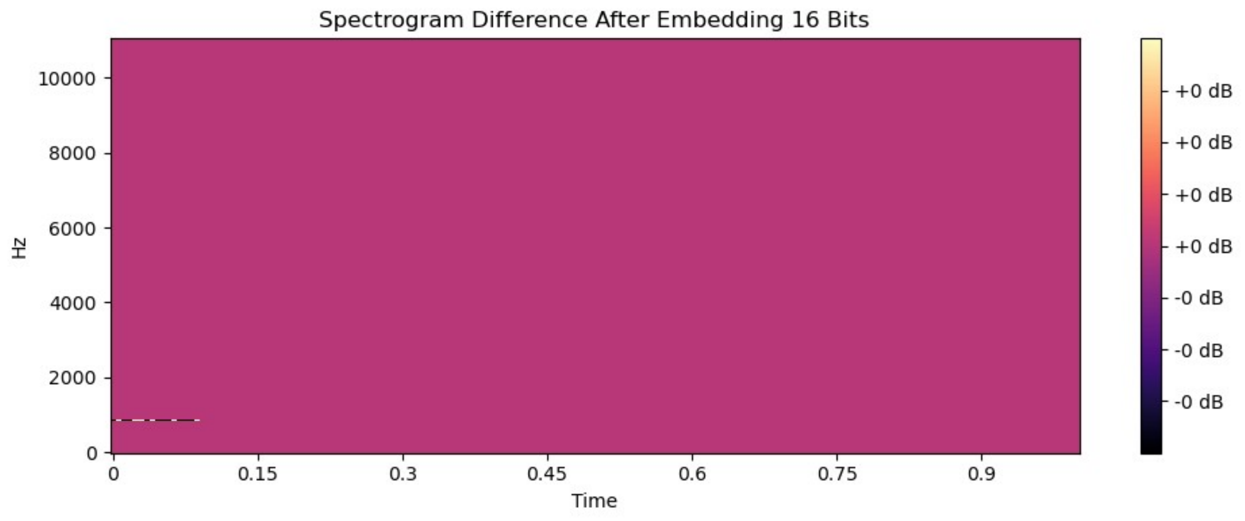
```
Total chunks created: 1
```

```
from pydub import AudioSegment

# Load audio and slice two overlapping 1s segments
audio = AudioSegment.from_file("fur_elise.mp3").set_channels(1)
audio_1 = audio[10000:11000]  # 10s to 11s
audio_2 = audio[10500:11500]  # 10.5s to 11.5s (overlaps with audio_1)

# Embed the same chunk in both
embedded_1 = {
    "audio_raw": audio_1.get_array_of_samples(),
    "frame_rate": audio_1.frame_rate,
    "chunks": [chunks[0]]  # only one chunk
}
```

```
embedded_2 = {
    "audio_raw": audio_2.get_array_of_samples(),
    "frame_rate": audio_2.frame_rate,
    "chunks": [chunks[0]]  # same chunk again
}

corrupted_chunks_1 = [bytes(chunk) for chunk in embedded_1["chunks"]]
corrupted_chunks_1[0] =
b"CORRUPTED_DATA".ljust(len(corrupted_chunks_1[0]), b'\0')

# Keep the second segment's chunks clean
clean_chunks_2 = embedded_2["chunks"]

def recover_and_rebuild(chunks1, chunks2):
    print(" Starting recovery...")
    print(f" chunks1 count: {len(chunks1)}")
    print(f" chunks2 count: {len(chunks2)}")

    combined = chunks2 + chunks1  # prefer clean first
    seen_hashes = set()
    final = []

    for i, chunk in enumerate(combined):
        if chunk.startswith(b"CORRUPTED_DATA"):
            print(f" Skipping corrupted chunk at combined index {i}")
            continue
        h = hash(chunk)
        if h not in seen_hashes:
            seen_hashes.add(h)
            final.append(chunk)
        else:
            print(f" Duplicate chunk at index {i}, skipped")

    print(f" Final chunk count after deduplication: {len(final)}")

    recovered_bytes = b"".join(final)
    print(f" Total bytes to decompress: {len(recovered_bytes)}")

    try:
        decompressed = zlib.decompress(recovered_bytes)
        print(" Decompression succeeded")
        fields = decompressed.decode('utf-8').split('|')
        return {f.split(':')[0]: f.split(':')[1] for f in fields if
':' in f}
    except Exception as e:
        print(" Recovery failed:", e)
        return None
```

```
recovered = recover_and_rebuild(corrupted_chunks_1, clean_chunks_2)
print("\nRecovered Metadata:")
print(recovered)

□ Starting recovery...
□ chunks1 count: 1
□ chunks2 count: 1
⚠ Skipping corrupted chunk at combined index 1
□ Final chunk count after deduplication: 1
□ Total bytes to decompress: 300
□ Decompression succeeded

Recovered Metadata:
{'ISRC': 'TEST99999999', 'Title': 'Für Elise', 'Performer':
'Beethoven', 'Duration': '05', 'Owner': 'Public Domain'}

# Reuses pre-compressed metadata chunks, embeds them into two
overlapping segments
# with redundancy (including chunk A in both), introduces corruption,
and attempts recovery.

import numpy as np
import random
import zlib
from pydub import AudioSegment

# Embed Chunks into Two Overlapping Audio Segments
def embed_chunks_in_audio(audio_segment_1, audio_segment_2, chunks):
    embedded_1 = {
        "audio_raw": audio_segment_1.get_array_of_samples(),
        "frame_rate": audio_segment_1.frame_rate,
        "chunks": chunks[0:4]  # A, B, C, D
    }
    embedded_2 = {
        "audio_raw": audio_segment_2.get_array_of_samples(),
        "frame_rate": audio_segment_2.frame_rate,
        "chunks": chunks[0:4]  # A, B, C, D (again for redundancy)
    }
    return embedded_1, embedded_2

# Corrupt a Random Chunk
def corrupt_random_chunk(embedded):
    chunks = embedded["chunks"]
    corrupted = [bytes(chunk) for chunk in chunks]  # Deep copy!
    index = random.randint(0, len(corrupted) - 1)
    corrupted[index] = b"CORRUPTED_DATA".ljust(len(corrupted[index]),
b'\0')
    print(f"Corrupted chunk index in this slice: {index}")
    return corrupted
```

```
# Rebuild Metadata from Combined Chunks
def recover_and_rebuild(chunks1, chunks2):
    combined = chunks1 + chunks2
    seen = set()
    final = []
    for chunk in combined:
        if chunk not in seen:
            seen.add(chunk)
            final.append(chunk)

    recovered_bytes = b"".join(final)
    try:
        decompressed = zlib.decompress(recovered_bytes)
        fields = decompressed.decode('utf-8').split('|')
        return {f.split(':')[0]: f.split(':')[1] for f in fields if
':' in f}
    except Exception as e:
        print("Recovery failed:", e)
        return None

# Step 1: Embed a 16-bit micro-chunk into the spectrogram of a 1-
second audio segment

import librosa
import librosa.display
import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf

# Load 1 second of audio
one_sec_audio, sr = librosa.load("fur_elise.mp3", sr=22050, offset=10,
duration=1.0)

# Compute STFT
D = librosa.stft(one_sec_audio, n_fft=512, hop_length=128)
mag, phase = np.abs(D), np.angle(D)

# Simulate micro-chunk as 16 bits for demo purposes
micro_chunk = "1011001011101110"  # 16 bits

# Embed bits into the magnitude of a stable frequency band say row 20
mag_embedded = mag.copy()
row = 20  # frequency bin index

for i, bit in enumerate(micro_chunk):
    col = i  # each bit goes in a new frame
    if col < mag.shape[1]:
        if bit == '1':
            mag_embedded[row, col] += 0.01  # boost a little
        else:
            mag_embedded[row, col] -= 0.01  # suppress a little

# Reconstruct complex spectrogram and inverse STFT
D_embedded = mag_embedded * np.exp(1j * phase)
embedded_audio = librosa.istft(D_embedded, hop_length=128)

# Save embedded audio to file
sf.write("fur_elise_embedded.wav", embedded_audio, sr)

# Plot difference
plt.figure(figsize=(10, 4))
librosa.display.specshow(mag - mag_embedded, sr=sr, hop_length=128,
x_axis='time', y_axis='linear')
plt.colorbar(format="%+2.0f dB")
plt.title("Spectrogram Difference After Embedding 16 Bits")
plt.tight_layout()
plt.show()
```



```
# Reuses pre-compressed metadata chunks, embeds them into two
overlapping segments
# with redundancy (including chunk A in both), introduces corruption,
and attempts recovery.

# Reconstruct complex spectrogram and inverse STFT
D_embedded = mag_embedded * np.exp(1j * phase)
embedded_audio = librosa.istft(D_embedded, hop_length=128)

# Save embedded audio to file
sf.write("fur_elise_embedded.wav", embedded_audio, sr)

# Recovery Step
# Load embedded audio and extract back the 16 bits
embedded_loaded, _ = librosa.load("fur_elise_embedded.wav", sr=22050,
duration=1.0)
D_loaded = librosa.stft(embedded_loaded, n_fft=512, hop_length=128)
mag_loaded = np.abs(D_loaded)

recovered_bits = ""
for i in range(16):
    col = i
    if col < mag_loaded.shape[1]:
        delta = mag_loaded[row, col] - mag[row, col]
        recovered_bits += '1' if delta > 0 else '0'

print("\n Recovered Bits:", recovered_bits)


 Recovered Bits: 1011001011101110
```