Research for Blueprint 1

“ # SSIS Research Pack v1.0

SOVRN Streaming Intelligence Spine — Consolidated Research for Implementation

Date: January 2026

Purpose: Ground truth research to inform Blueprint #1 (Audio Pipeline) implementation

---

Executive Summary

This document consolidates research across 5 domains to enable confident implementation of the SSIS Audio Pipeline. Key decisions are locked; open questions are flagged for empirical testing.

Locked Decisions

Component	Decision	Rationale
Orchestrator	Huey + SQLite	Offline-first, zero infrastructure, Python-native
Embedding Model	YAMNet (v1)	3.7M params, MIT license, real-time CPU, 1024-D output
Segmentation	inaSpeechSegmenter	MIT, speech/music/noise out-of-box, MIREX 2018 winner
Feature Set	Mel spectrogram + YAMNet embedding	Dual-scale: 10ms hop (mel), 0.5s hop (embedding)
Storage Format	HDF5 with gzip compression	Random access, mature tooling, ~300KB/min audio
Preview Selection	Heuristic v1	Sentence breaks + energy scoring, fallback to intro
Observability	SQLite + JSON logs	Portable, offline-first, simple dashboard

---

Domain 1: Pipeline Orchestration & Infrastructure

Framework Selection

Recommendation: Huey + SQLite

Huey provides:

- Task queue with clean Python API

- SQLite, Redis, file-system, or in-memory storage backends

- Retries, scheduling, task prioritization, result storage

- Lightweight enough for offline-first deployment

```python
from huey import SqliteHuey

huey = SqliteHuey('ssis-pipeline', filename='ssis_queue.db')

@huey.task(retries=3, retry_delay=60)
def process_audio(asset_id: str):
    # Pipeline stages here
    pass
```

Migration path: If durable execution becomes critical (e.g., multi-hour jobs surviving power outages), migrate to Temporal. Temporal makes code durable—workflows survive server crashes, network partitions, and week-long delays.

Idempotency Pattern (Stolen from Klio)

```python
def should_process(asset_id: str, stage: str) -> bool:
    """Check if output already exists before processing."""
    output_path = get_output_path(asset_id, stage)
    if output_path.exists():
        log.info(f"Skipping {asset_id}/{stage}: output exists")
        return False
    return True

def process_stage(asset_id: str, stage: str, processor_fn):
    if not should_process(asset_id, stage):
        return  # Idempotent skip

    result = processor_fn(asset_id)
    save_output(asset_id, stage, result)
    emit_event(f"{stage}.ready", asset_id)
```

Failure Recovery

- **Retry with exponential backoff:** 3 attempts, delays of 60s, 300s, 900s

- **Dead letter after N failures:** Log to `failed_jobs` table for manual review

- **Checkpointing for long files:** Write intermediate results every 60s of audio processed

- **Atomic writes:** Write to temp file, then rename (survives power loss)

Load-Shedding Resilience

For South African infrastructure constraints:

- Process in small chunks (30-60s segments)

- Checkpoint after each chunk

- Resume from last successful chunk on restart

- Store queue state in SQLite (survives restart)

Domain 2: Audio Feature Engineering

Embedding Model Selection

v1: YAMNet

Attribute	Value
Architecture	MobileNetV1 (depthwise separable conv)
Parameters	~3.7M
Embedding Dimension	1024-D
Training Data	AudioSet (521 classes)
CPU Performance	Real-time or faster
License	MIT

Why YAMNet over alternatives:

- VGGish (60M params) is heavier with smaller 128-D output

- PANNs CNN14 (80M params) is more accurate but ~2x slower

- CLAP/MERT require GPU for reasonable performance

Upgrade path:

- **PANNs CNN14** for improved accuracy (when CPU budget allows)

- **CLAP** for semantic search ("find clips sounding like X")

- **OpenL3** for music-specific similarity

Minimum Viable Feature Set

Feature	Specification	Purpose
Log-Mel Spectrogram	64 mel bands, 10ms hop, 25ms window	Segmentation, visualization, input to models
YAMNet Embedding	1024-D vector, every 0.5-1s	Classification, similarity search, preview selection
RMS Energy	Per-frame (10ms)	Preview selection, dynamic detection

Storage estimate per minute of audio:

- Mel: 6000 frames × 64 bands × 4 bytes = 1.5 MB → ~200KB compressed

- Embeddings: 120 vectors × 1024 × 4 bytes = 480 KB → ~100KB compressed

- **Total: ~300-400 KB/minute** (vs ~10 MB/minute raw audio)

Frame and Hop Size Strategy

Dual-scale processing:

```python
# For segmentation (high temporal resolution)
mel_config = {
    'n_mels': 64,
    'hop_length': 220,      # 10ms at 22050 Hz
    'win_length': 551,      # 25ms window
    'n_fft': 1024,
    'sample_rate': 22050
}
# For embeddings (sufficient context)
embedding_config = {
    'window_sec': 1.0,      # 1 second context per embedding
    'hop_sec': 0.5,         # 50% overlap
    'model': 'yamnet'
}
```

Rationale: Segmentation needs ~10ms resolution to catch boundaries. Embeddings need ~1s context for semantic meaning. Process mel at high resolution; compute embeddings on larger windows.

Feature Storage Format

Recommendation: HDF5 with gzip compression

```python
import h5py

def save_feature_pack(asset_id: str, mel: np.ndarray, embeddings: np.ndarray):

    with h5py.File(f'features/{asset_id}.h5', 'w') as f:

        f.create_dataset('mel', data=mel, compression='gzip', compression_opts=4)

        f.create_dataset('embeddings', data=embeddings, compression='gzip', compression_opts=4)

        f.attrs['sample_rate'] = 22050

        f.attrs['mel_hop'] = 220

        f.attrs['embedding_hop_sec'] = 0.5

        f.attrs['version'] = '1.0'
```

Why HDF5:

- Random access (load specific time range without reading whole file)

- Good compression (4-5x on mel spectrograms)

- Mature tooling (h5py)

- Single file per asset (easy to manage)

Alternative for simplicity: NPZ per file (simpler but no random access)

Future consideration: Lance for vector search at scale

CPU Optimization Strategies

1. **Model loading:** Load YAMNet once, reuse for all files

2. **Batching:** Process multiple windows in single forward pass

3. **Threading:** Set `OMP_NUM_THREADS` appropriately

4. **ONNX export:** Convert YAMNet to ONNX for optimized CPU inference

5. **Quantization:** INT8 quantization for ~2x speedup (test accuracy first)

Domain 3: Speech/Music Segmentation

v1 Segmentation Stack

Primary: inaSpeechSegmenter

```python
```

from inaSpeechSegmenter import Segmenter

seg = Segmenter(vad_engine='smn', detect_gender=False)

result = seg('audio.wav')

Returns: [('speech', 0.0, 2.5), ('music', 2.5, 5.0), ('noise', 5.0, 6.0), ...]

```

| Attribute | Value |
|-----------|-------|
| Classes | speech, music, noise (+ optional gender) |
| Architecture | CNN (4 conv + 4 dense) + Viterbi smoothing |
| Mel Bands | 21 (100-4000 Hz range) |
| License | MIT |
| Accuracy | ~96% F1 (MIREX 2018 winner) |

**Preprocessing: Silero VAD**

```python
from silero_vad import load_silero_vad, get_speech_timestamps

vad_model = load_silero_vad()

speech_timestamps = get_speech_timestamps(wav, vad_model, threshold=0.5)
```

- Ultra-fast: <1ms per 30ms chunk
- Tiny model: ~2MB
- Use for quick silence detection before expensive segmentation

Post-Processing Requirements

```python
def post_process_segments(segments: list) -> list:

    # 1. Minimum duration filtering

    segments = filter_short_segments(

        segments,

        min_speech=0.8,    # seconds

        min_music=3.4,     # seconds

        min_silence=0.5    # seconds
```
```

```
    )

    # 2. Merge adjacent same-class segments
    segments = merge_adjacent(segments, gap_threshold=0.3)

    # 3. Apply hysteresis for stable boundaries
    segments = apply_hysteresis(segments, onset=0.6, offset=0.4)

    return segments
```

Heuristic Fallback (When ML Confidence Low)

Signal processing features achieving 94%+ accuracy:

Feature	Accuracy	Computation
Spectral flux variance	94%	L2-norm of frame diff, variance over 1s
4Hz modulation energy	88%	Bandpass on energy envelope (speech syllable rate)
Spectral centroid variance	86%	Center-of-mass variance over 1s
Low-energy frame %	86%	Frames below 50% mean energy

```python
def heuristic_speech_music(audio: np.ndarray, sr: int) -> str:
    """Fallback when ML confidence < 0.5"""
    mod_4hz = compute_4hz_modulation(audio, sr)
    spectral_flux_var = compute_spectral_flux_variance(audio, sr)

    # Speech has higher 4Hz modulation (syllable rate)
    if mod_4hz > 0.3 and spectral_flux_var > threshold:
        return 'speech'
    else:
        return 'music'
```

Confidence Calibration

```python
# Temperature scaling for calibrated probabilities
def calibrate_confidence(logits: np.ndarray, temperature: float = 2.0) -> np.ndarray:
    return softmax(logits / temperature)
# Production thresholds
CONFIDENCE_THRESHOLDS = {
    'accept': 0.8,      # Use prediction directly
    'review': 0.5,      # Use but flag for batch QA
    'fallback': 0.3,    # Trigger heuristic fallback
    'uncertain': 0.0    # Mark as uncertain
}
```

Known Limitations & Guards

Issue	Mitigation
Crashes on files < 1.7s	Check duration before processing
Memory issues on very long files	Process in chunks (5-10 min)
Speech-over-music classified as speech	Accept for v1; hierarchical detection for v2
Rap can confuse speech/music	Use pitch stability features

v2 Weak Supervision Path

1. **Generate weak labels:**
   - Whisper transcripts → speech regions
   - Audio fingerprinting (AcoustID) → music regions
   - Chapter markers → structural hints
2. **Combine with Snorkel:**

```python
@labeling_function()
def lf_whisper_confident(x):
    if x.whisper_confidence > 0.8 and len(x.transcript) > 0:
```

```
        return SPEECH
    return ABSTAIN

@labeling_function()
def lf_4hz_modulation(x):
    if x.modulation_4hz > 0.3:
        return SPEECH
    return ABSTAIN
```

3. **Train CRNN on noisy labels:**

   - Architecture: 3 conv + 2 bi-GRU (Netflix SMAD: 831k params)

   - Loss: Symmetric Cross-Entropy (robust to 40%+ label noise)

   - Post-training: Cleanlab for label cleaning

Domain 4: Preview Selection Heuristics

Spotify's Approach (Reference)

Their legacy system:

1. Detect sentence boundaries (sentence starts/ends as preview candidates)

2. Score candidates with "selectivity" model

3. Trim to improve coherence (respect sentence boundaries)

4. Rank and select highest-scoring candidate

5. Fallback: episode intro if no good candidates

Their LLM upgrade achieved 4.6% engagement increase + 5x processing efficiency.

SSIS v1 Preview Algorithm

```python
def select_preview(
    audio: np.ndarray,
    segments: list,
    embeddings: np.ndarray,
    duration: float = 60.0
) -> dict:
```

```
    """Select best preview clip from audio."""

    # 1. Find speech-heavy regions (for podcasts)
    speech_regions = [s for s in segments if s['label'] == 'speech']

    # 2. Detect sentence boundaries using energy + pause detection
    boundaries = detect_sentence_boundaries(audio, segments)

    # 3. Generate candidate clips at boundaries
    candidates = generate_candidates(boundaries, target_duration=duration)

    # 4. Score each candidate
    for candidate in candidates:
        candidate['score'] = compute_engagement_score(
            audio[candidate['start']:candidate['end']],
            embeddings[candidate['start_idx']:candidate['end_idx']]
        )

    # 5. Select best or fallback
    best = max(candidates, key=lambda x: x['score'])

    if best['score'] < CONFIDENCE_THRESHOLD:
        return fallback_to_intro(audio, duration)

    return best
def compute_engagement_score(audio_clip: np.ndarray, embeddings: np.ndarray) -> float:
    """Score based on energy variance and speech presence."""

    # Energy variance (dynamic = interesting)
    rms = librosa.feature.rms(y=audio_clip)[0]
```

```
    energy_variance = np.var(rms)

    # Embedding diversity (not monotonous)
    embedding_variance = np.mean(np.var(embeddings, axis=0))

    # Combine scores (tune weights empirically)
    return 0.6 * normalize(energy_variance) + 0.4 * normalize(embedding_variance)
```

Sentence Boundary Detection

```python
def detect_sentence_boundaries(audio: np.ndarray, sr: int) -> list:
    """Find natural speech boundaries using energy and pauses."""

    # Compute energy
    rms = librosa.feature.rms(y=audio, hop_length=512)[0]

    # Find pauses (low energy regions > 200ms)
    threshold = np.mean(rms) * 0.3
    is_pause = rms < threshold

    # Find pause boundaries
    boundaries = []
    in_pause = False
    pause_start = 0

    for i, pause in enumerate(is_pause):
        if pause and not in_pause:
            pause_start = i
            in_pause = True
        elif not pause and in_pause:
```

```
            pause_duration = (i - pause_start) * 512 / sr
            if pause_duration > 0.2:  # 200ms minimum pause
                boundaries.append({
                    'time': pause_start * 512 / sr,
                    'type': 'pause_end'
                })
            in_pause = False

    return boundaries
```

Fallback Hierarchy

1. **Smart selection** → best-scoring candidate with confidence > 0.5
2. **Intro** → first 60s after any leading silence
3. **First segment** → literally start of content

Domain 5: Observability

Metrics to Track

Per-stage metrics:

Stage	Metrics
Ingest	`files_ingested`, `validation_failures`, `codec_distribution`
Decode	`decode_duration_ratio`, `format_errors`, `sample_rate_mismatches`
Features	`extraction_time_ms`, `nan_inf_count`, `dimension_checks`
Segments	`segment_count`, `low_confidence_ratio`, `class_distribution`
Preview	`fallback_rate`, `confidence_histogram`, `selection_time_ms`

Pipeline-level metrics:

- `end_to_end_latency_ms` (p50, p95, p99)
- `success_rate` (target: > 95%)
- `backlog_depth`
- `throughput_files_per_hour`

Logging Schema

```python
@dataclass
class PipelineJobLog:
    job_id: str
    asset_id: str
    stage: str  # ingest | decode | features | segments | preview
    status: str  # running | success | failed | skipped
    attempt: int
    started_at: datetime
    ended_at: datetime
    duration_ms: int
    error_code: Optional[str]
    error_message: Optional[str]
    metrics: dict  # stage-specific metrics
```

```sql
CREATE TABLE pipeline_jobs (
    job_id TEXT PRIMARY KEY,
    asset_id TEXT NOT NULL,
    stage TEXT NOT NULL,
    status TEXT NOT NULL,
    attempt INTEGER DEFAULT 1,
    started_at TEXT NOT NULL,
    ended_at TEXT,
    duration_ms INTEGER,
    error_code TEXT,
    error_message TEXT,
    metrics_json TEXT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
```

);

CREATE INDEX idx_asset_stage ON pipeline_jobs(asset_id, stage);

CREATE INDEX idx_status ON pipeline_jobs(status);

```

### Error Taxonomy

| Error Code | Category | Description | Action |
|------------|----------|-------------|--------|
| `CODEC_UNSUPPORTED` | Decode | Unknown audio format | Skip, log for review |
| `FILE_CORRUPT` | Decode | Cannot read file | Skip, alert if frequent |
| `FILE_TOO_SHORT` | Validation | Duration < 1.7s | Skip (inaSpeechSegmenter limit) |
| `FEATURE_NAN` | Features | NaN/Inf in output | Retry with different params |
| `MODEL_OOM` | Features | Out of memory | Reduce chunk size, retry |
| `SEGMENTATION_FAILED` | Segments | No segments produced | Use heuristic fallback |
| `PREVIEW_LOW_CONF` | Preview | All candidates below threshold | Use intro fallback |

### Alerting Rules

```python
ALERTS = {
    'success_rate_low': {
        'condition': 'success_rate < 0.95 over 1 hour',
        'severity': 'warning'
    },
    'backlog_growing': {
        'condition': 'backlog_depth increasing for > 30 min',
        'severity': 'warning'
    },
    'repeated_failures': {
        'condition': 'same asset_id failed > 3 times',
        'severity': 'error'
    },
    'latency_regression': {

        'condition': 'p95_latency > 2x baseline',
        'severity': 'warning'
    }
}
```

Simple Dashboard (v1)

Track in SQLite, visualize with any tool:

```python
def get_pipeline_health() -> dict:
    """Quick health check for dashboard."""


    last_hour = datetime.now() - timedelta(hours=1)


    return {
        'success_rate': db.query('''
            SELECT 1.0 * SUM(CASE WHEN status='success' THEN 1 ELSE 0 END) / COUNT(*)
            FROM pipeline_jobs WHERE started_at > ?
        ''', [last_hour]),


        'avg_latency_ms': db.query('''
            SELECT AVG(duration_ms) FROM pipeline_jobs
            WHERE status='success' AND started_at > ?
        ''', [last_hour]),


        'backlog_depth': queue.pending_count(),


        'error_breakdown': db.query('''
            SELECT error_code, COUNT(*) FROM pipeline_jobs
            WHERE status='failed' AND started_at > ?
            GROUP BY error_code

```
        ''', [last_hour])
    }
```

Data Contracts (JSON Schema)

AudioAsset

```json
{
  "asset_id": "uuid",
  "source": {
    "type": "upload|url|local",
    "uri": "string"
  },
  "owner_entity_id": "uuid",
  "created_at": "iso8601",
  "duration_sec": 0.0,
  "channels": 1,
  "sample_rate_hz": 22050,
  "format": "wav|mp3|flac|m4a",
  "content_hash": "sha256",
  "status": "registered|processing|ready|failed"
}
```

FeaturePack

```json
{
  "asset_id": "uuid",
  "version": "1.0",
  "mel": {
    "uri": "features/{asset_id}.h5#mel",
```

```
    "shape": [6000, 64],
    "dtype": "float32",
    "hop_ms": 10,
    "n_mels": 64
  },
  "embeddings": {
    "uri": "features/{asset_id}.h5#embeddings",
    "shape": [120, 1024],
    "dtype": "float32",
    "hop_sec": 0.5,
    "model": "yamnet"
  },
  "computed_at": "iso8601"
}
```

Segments

```json
{
  "asset_id": "uuid",
  "version": "1.0",
  "segments": [
    {
      "start_sec": 0.0,
      "end_sec": 2.5,
      "label": "speech|music|noise|silence",
      "confidence": 0.95
    }
  ],
  "model": "inaSpeechSegmenter",
  "model_version": "0.8.0",
```

  "computed_at": "iso8601"

}

```

### PreviewCandidate

```json
{
  "asset_id": "uuid",
  "mode": "smart|intro|fallback",
  "start_sec": 45.0,
  "end_sec": 105.0,
  "duration_sec": 60.0,
  "confidence": 0.78,
  "fallback_used": false,
  "reason": "highest_engagement_score",
  "computed_at": "iso8601"
}
```

Implementation Roadmap

MVP Scope (Week 1-2)

Must ship:

1. ✅ Ingest audio (local file path)

2. ✅ Decode to canonical WAV (22050 Hz mono)

3. ✅ Extract mel spectrogram (64 bands, 10ms hop)

4. ✅ Extract YAMNet embeddings (1024-D, 0.5s hop)

5. ✅ Run inaSpeechSegmenter (speech/music/noise)

6. ✅ Generate preview offset (heuristic + fallback)

7. ✅ Save FeaturePack to HDF5

8. ✅ Log all jobs to SQLite

Post-MVP (Week 3-4)
```

- QC stats (loudness, clipping, silence ratio)

- Batch processing CLI

- Simple health dashboard

- Resume-from-checkpoint for interrupted jobs

v2 Features (Future)

- Weak supervision training pipeline

- CLAP integration for semantic search

- API endpoint for feature retrieval

- Vector similarity search

Open Questions (Require Empirical Testing)

1. **YAMNet accuracy on SA content:** Does YAMNet distinguish speech/music well on South African media (accents, local music styles)?

2. **Preview engagement:** Do automatically selected previews match human preferences? Need A/B testing framework.

3. **Optimal confidence thresholds:** Are 0.8/0.5/0.3 thresholds correct for this data? Calibrate on validation set.

4. **Storage format performance:** Is HDF5 fast enough for random access patterns? Test vs NPZ vs Lance.

5. **Power resilience:** Does checkpoint-every-60s provide adequate protection? Simulate power cuts.

Dependencies

```

# requirements.txt

huey>=2.5.0

h5py>=3.10.0

numpy>=1.24.0

librosa>=0.10.0

torch>=2.0.0

torchaudio>=2.0.0

tensorflow-hub>=0.15.0  # for YAMNet

inaSpeechSegmenter>=0.8.0
```

silero-vad>=5.0

soundfile>=0.12.0

```

---

## Next Steps for ChatGPT

Use this research pack to generate:

1. **Implementation prompts** for each pipeline stage

2. **Test specifications** for contract validation

3. **Codegen for data schemas** (SQLAlchemy models, Pydantic schemas)

4. **CLI interface** for batch processing

5. **Docker configuration** for reproducible environment

The key constraint: **offline-first, CPU-only, South African infrastructure resilience**.

---

*Research compiled from: Spotify Engineering Blog, Netflix TechBlog, ArXiv papers (CLAP, MERT, SMAD), tool documentation (inaSpeechSegmenter, YAMNet, Huey, HDF5), and web research on orchestration patterns and observability.*"

Here is the SSIS Research Pack containing consolidated research for building an audio processing pipeline. Based on this document:

1. Generate implementation prompts for each pipeline stage (ingest, decode, features, segments, preview)

2. Create the folder structure and interface definitions

3. Write the first implementation: the Ingest API + schemas + contract tests

Follow the locked decisions exactly. The key constraints are: offline-first, CPU-only, Python/PyTorch stack, South African infrastructure resilience.

## Small adjustments I recommend

1. YAMNet vs "PyTorch stack"

   YAMNet is most commonly consumed via TensorFlow Hub. That's fine for SSIS (it's still Python + CPU), but it's not "pure PyTorch." Two safe paths:

   - v1: allow TFHub only inside worker_features (keep the rest PyTorch/CPU).

   - v1.1: export YAMNet to ONNX and run with onnxruntime CPU.
```

2. inaSpeechSegmenter labels

   It returns speech/music/noise. Your contract includes silence. That’s fine: we’ll derive silence from gaps + optional Silero VAD.

3. Huey + SQLite concurrency

   SQLite-backed queues are best with one consumer process (or careful locking). We’ll design v1 as: 1 worker per machine, chunking/checkpoints for resilience.

4. HDF5 writes

   HDF5 is great, but concurrent writes are tricky. We’ll structure it so only worker_features writes the .h5, and later stages only write JSON sidecars.

5. Sample rate

   You locked 22050 Hz mono; we’ll align all contracts + decode normalization to that.