Fish Market Voice Processing Technical Architecture

System Overview

High-Level Architecture
```
[Phone Microphone] → [Audio Preprocessing] → [Voice Activity Detection]
                                    ↓
[Transaction Extraction] ← [Speaker Identification] ← [Speech-to-Text]
                                    ↓
[Confidence Scoring] → [Local Database] → [Daily Summary] → [User Interface]
```

Core Design Principles
1. **On-Device Processing**: Privacy and offline capability
2. **Low Battery Usage**: Optimized for all-day listening
3. **Noise Resilience**: Works in busy market environments
4. **Real-time Processing**: Minimal delay between speech and recognition
5. **Graceful Degradation**: Works even with poor audio quality

Detailed Component Architecture

1. Audio Capture & Preprocessing Layer

Audio Input Manager
```kotlin
// Android Implementation Example
class AudioInputManager {
    private val audioRecord: AudioRecord
    private val bufferSize: Int = 8192
    private val sampleRate: Int = 16000 // Optimized for speech

    // Continuous audio capture with circular buffer
    fun startListening() {
        // Use low-latency audio capture
        // Implement noise gate to ignore very quiet audio
```

```
        // Circular buffer to keep last 30 seconds
    }

    // Smart listening - only process when likely transaction
    fun detectSpeechActivity(): Boolean {
        // Use volume threshold + frequency analysis
        // Detect multiple speakers (seller + customer)
        // Filter out pure background noise
    }
}
```

Audio Preprocessing Pipeline
```
Raw Audio (16kHz, 16-bit)
    ↓
Noise Reduction (Spectral Subtraction)
    ↓
Voice Activity Detection (VAD)
    ↓
Speaker Separation (Basic 2-speaker model)
    ↓
Audio Segmentation (Transaction chunks)
    ↓
Feature Extraction (MFCC coefficients)
```

Key Technologies:
- **WebRTC VAD**: Excellent voice activity detection
- **RNNoise**: Real-time noise suppression optimized for mobile
- **Librosa/TensorFlow**: Audio feature extraction

2. Speech Recognition Engine

Multi-Stage Recognition Pipeline
```python
class FishMarketSTT:
```

```
    def __init__(self):
        # Stage 1: General speech recognition
        self.general_stt = WhisperModel("base.en")

        # Stage 2: Fish market specific model
        self.fish_stt = CustomModel("fish_market_v1.tflite")

        # Stage 3: Local language components
        self.twi_stt = LocalLanguageModel("twi_basic.tflite")

    def process_audio(self, audio_chunk):
        # Parallel processing for speed
        general_text = self.general_stt.transcribe(audio_chunk)
        fish_text = self.fish_stt.transcribe(audio_chunk)
        local_text = self.twi_stt.transcribe(audio_chunk)

        # Confidence weighted combination
        return self.combine_results(general_text, fish_text, local_text)
```

Custom Fish Market Model Training
```python
# Training Data Structure
training_samples = {
    "transaction_type": "price_inquiry",
    "audio_file": "sample_001.wav",
    "transcript": "How much be this tilapia?",
    "speaker": "customer",
    "confidence": 0.95,
    "fish_type": "tilapia",
    "price_mentioned": None,
    "language_mix": ["english", "pidgin"]
}

# Model Architecture
class FishMarketASR(tf.keras.Model):
    def __init__(self):
```

```
        # Transformer-based architecture optimized for:
        # - Code-switching (English + Twi)
        # - Fish-specific vocabulary
        # - Market noise resilience
        # - Low-latency inference
```

3. Transaction Detection & Parsing

Transaction State Machine
```python
class TransactionStateMachine:
    def __init__(self):
        self.state = "IDLE"
        self.current_transaction = {}
        self.confidence_threshold = 0.7

    def process_utterance(self, text, speaker, confidence):
        if self.state == "IDLE":
            if self.is_price_inquiry(text):
                self.state = "PRICE_INQUIRY"
                self.start_new_transaction(text, speaker)

        elif self.state == "PRICE_INQUIRY":
            if self.is_price_response(text, speaker):
                self.state = "PRICE_GIVEN"
                self.add_price_info(text)

        elif self.state == "PRICE_GIVEN":
            if self.is_payment_confirmation(text):
                self.state = "PAYMENT"
                self.complete_transaction()

        # Timeout handling - reset if no progress for 2 minutes
        self.check_timeout()

    def is_price_inquiry(self, text):
```

```
        patterns = [
            r"how much.*fish",
            r"what.*price",
            r"sɛn na ɛyɛ",
            r"how much be",
            r"price.*tilapia"
        ]
        return any(re.search(pattern, text.lower()) for pattern in patterns)
```

Entity Extraction Engine
```python
class FishMarketNER:
    def __init__(self):
        # Custom NER model for fish market entities
        self.fish_patterns = self.load_fish_patterns()
        self.price_patterns = self.load_price_patterns()
        self.quantity_patterns = self.load_quantity_patterns()

    def extract_entities(self, text):
        entities = {
            "fish_type": self.extract_fish_type(text),
            "price": self.extract_price(text),
            "quantity": self.extract_quantity(text),
            "currency": self.extract_currency(text)
        }
        return entities

    def extract_fish_type(self, text):
        # Pattern matching + fuzzy matching for fish names
        fish_keywords = {
            "tilapia": ["tilapia", "tuo", "apateshi"],
            "tuna": ["tuna", "light meat"],
            "mackerel": ["mackerel", "kpanla", "titus"],
            "sardine": ["sardine", "herring"]
        }
```

```
        # Use fuzzy string matching for variations
        from fuzzywuzzy import fuzz
        # Implementation details...
```

4. Speaker Identification System

Simple Speaker Diarization
```python
class SpeakerIdentifier:
    def __init__(self):
        # Lightweight speaker identification
        self.seller_voice_profile = None
        self.current_speakers = {}

    def learn_seller_voice(self, audio_samples):
        # Build seller voice profile during setup
        # Extract speaker embeddings (x-vectors)
        # Store characteristic features
        pass

    def identify_speaker(self, audio_chunk):
        # Real-time speaker identification
        # Return: "seller", "customer", or "unknown"
        embedding = self.extract_speaker_embedding(audio_chunk)

        if self.is_similar_to_seller(embedding):
            return "seller"
        elif self.is_new_speaker(embedding):
            return "customer"
        else:
            return "unknown"
```

5. Confidence Scoring & Quality Control

Multi-Dimensional Confidence Scoring

```python
class ConfidenceScorer:
    def score_transaction(self, transaction_data):
        scores = {
            "audio_quality": self.score_audio_quality(transaction_data.audio),
            "speech_clarity": self.score_speech_clarity(transaction_data.text),
            "transaction_completeness": self.score_completeness(transaction_data),
            "entity_confidence": self.score_entities(transaction_data.entities),
            "pattern_match": self.score_pattern_match(transaction_data)
        }

        # Weighted combination
        total_confidence = (
            scores["audio_quality"] * 0.2 +
            scores["speech_clarity"] * 0.25 +
            scores["transaction_completeness"] * 0.3 +
            scores["entity_confidence"] * 0.15 +
            scores["pattern_match"] * 0.1
        )

        return total_confidence, scores

    def should_auto_record(self, confidence):
        return confidence > 0.8

    def should_flag_for_review(self, confidence):
        return 0.5 < confidence <= 0.8
```

6. Data Storage & Management

Local Database Schema
```sql
-- Transactions table
CREATE TABLE transactions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
```

```
    fish_type TEXT,
    quantity INTEGER,
    unit_price REAL,
    total_amount REAL,
    currency TEXT DEFAULT 'GHS',
    confidence_score REAL,
    raw_audio_hash TEXT,
    transcript TEXT,
    status TEXT DEFAULT 'confirmed',
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Daily summaries table
CREATE TABLE daily_summaries (
    date DATE PRIMARY KEY,
    total_sales REAL,
    transaction_count INTEGER,
    top_fish_type TEXT,
    average_transaction REAL,
    confidence_avg REAL
);

-- Audio metadata (for debugging/improvement)
CREATE TABLE audio_logs (
    id INTEGER PRIMARY KEY,
    timestamp DATETIME,
    duration_seconds REAL,
    noise_level REAL,
    speaker_count INTEGER,
    processed BOOLEAN DEFAULT FALSE
);
```

Data Synchronization Strategy
```python
class DataSyncManager:
    def __init__(self):
```

```
        self.local_db = SQLiteDatabase("fish_ledger.db")
        self.cloud_backup = FirebaseDatabase()
        self.sync_interval = 3600  # 1 hour

    def sync_to_cloud(self):
        # Upload only transaction summaries, not raw audio
        # Encrypt sensitive data
        # Handle offline/online transitions
        pending_transactions = self.local_db.get_unsynced()

        for transaction in pending_transactions:
            # Remove audio data before upload
            clean_transaction = self.remove_audio_data(transaction)
            self.cloud_backup.save(clean_transaction)
```

7. Mobile App Integration Architecture

Flutter App Structure
```dart
// Main app architecture
class FishLedgerApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BlocProvider(
        create: (context) => VoiceProcessingBloc(),
        child: MainScreen(),
      ),
    );
  }
}

// Voice processing service
class VoiceProcessingService {
  static const platform = MethodChannel('fish_ledger/voice');
```

```
  Future<void> startListening() async {
    // Start native audio processing
    await platform.invokeMethod('startVoiceProcessing');
  }

  Stream<Transaction> get transactionStream {
    // Stream of detected transactions
    return platform.invokeMethod('getTransactionStream');
  }
}
```

Background Processing
```kotlin
// Android background service
class VoiceProcessingService : Service() {
    private lateinit var audioProcessor: AudioProcessor
    private lateinit var transactionDetector: TransactionDetector

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        // Start foreground service for continuous listening
        startForeground(NOTIFICATION_ID, createNotification())

        // Initialize audio processing pipeline
        audioProcessor.startListening { audioChunk ->
            transactionDetector.process(audioChunk)
        }

        return START_STICKY // Restart if killed
    }

    private fun createNotification(): Notification {
        // Show subtle notification that app is listening
        return NotificationCompat.Builder(this, CHANNEL_ID)
            .setContentTitle("Fish Ledger Active")
            .setContentText("Listening for sales...")
            .setSmallIcon(R.drawable.ic_mic)
```

```
            .build()
    }
}
```

Performance Optimization

Battery Life Optimization
```python
class PowerManager:
    def __init__(self):
        self.listening_mode = "SMART"  # FULL, SMART, MINIMAL
        self.market_hours = (6, 18)  # 6 AM to 6 PM
        self.activity_threshold = 0.3

    def should_listen_actively(self):
        current_hour = datetime.now().hour

        # Only active listening during market hours
        if not (self.market_hours[0] <= current_hour <= self.market_hours[1]):
            return False

        # Reduce activity if phone is in pocket (low motion)
        if self.get_phone_activity() < self.activity_threshold:
            return False

        return True

    def adjust_processing_intensity(self):
        battery_level = self.get_battery_level()

        if battery_level < 20:
            return "MINIMAL"  # Basic detection only
        elif battery_level < 50:
            return "SMART"    # Optimized processing
        else:
            return "FULL"     # All features active
```

```

### **Memory Management**
```python
class MemoryManager:
    def __init__(self):
        self.max_audio_buffer = 30  # seconds
        self.max_transactions_cache = 100
        self.cleanup_interval = 300  # 5 minutes

    def manage_audio_buffer(self):
        # Keep only recent audio for context
        # Delete processed audio older than 30 seconds
        # Compress older audio if needed for debugging
        pass

    def cleanup_old_data(self):
        # Remove old temporary files
        # Compress old transaction data
        # Clear ML model caches
        pass
```

Development Implementation Plan

Phase 1: Core Processing Engine (Months 1-2)
```
Priority Components:
1. Basic audio capture and preprocessing
2. Simple speech-to-text (English only)
3. Pattern matching for price/fish keywords
4. Local SQLite database
5. Basic confidence scoring

Technology Stack:
- Flutter for mobile app
- TensorFlow Lite for on-device ML
```
```

- SQLite for local storage
- WebRTC for audio processing
```

### **Phase 2: Enhanced Recognition (Months 3-4)**
```
Enhanced Components:
1. Multi-language support (Twi integration)
2. Speaker identification
3. Transaction state machine
4. Improved confidence scoring
5. Background service optimization

Additional Technologies:
- Custom trained models
- Advanced audio preprocessing
- Cloud backup integration
```

### **Phase 3: Production Ready (Months 5-6)**
```
Production Features:
1. Full error handling and recovery
2. Performance optimization
3. User feedback integration
4. Analytics and monitoring
5. Security and privacy features

Deployment Stack:
- Firebase for backend services
- Crashlytics for error tracking
- Analytics for usage monitoring
```

## Testing Strategy

### **Audio Testing Framework**
```

```python
class AudioTestSuite:
    def __init__(self):
        self.test_audio_samples = self.load_test_samples()
        self.ground_truth_transactions = self.load_ground_truth()

    def test_recognition_accuracy(self):
        # Test with various audio conditions
        results = {}

        for sample in self.test_audio_samples:
            predicted = self.voice_processor.process(sample.audio)
            expected = sample.ground_truth

            accuracy = self.calculate_accuracy(predicted, expected)
            results[sample.id] = accuracy

        return results

    def test_noise_resilience(self):
        # Add synthetic noise to clean samples
        # Test performance degradation
        pass
```

This architecture prioritizes Luther's principles: it's accessible (works on basic smartphones), practical (solves real problems), and focuses on authentic usage patterns (real market conversations). The key is starting simple and building complexity gradually based on real user feedback.