### config/settings.yaml

```yaml
audio:
  input:
    device_index: 1
    channels: 1
    sample_rate: 44100
    chunk_size: 4096
    format: "int16"
  output:
    device_index: 0
    sample_rate: 22050
    buffer_size: 2048
    channels: 1
  processing:
    noise_reduction: true
    auto_gain: true
    vad_aggressiveness: 2
    silence_threshold: 150
    silence_duration: 1.5
  wake_word:
    enabled: false
    keyword: "hey buddy"
    sensitivity: 0.5
# Pending implementation
vision:
  camera:
    resolution: [640, 480]
    framerate: 30
    rotation: 0
    hflip: false
    vflip: false
    exposure_mode: "auto"
    awb_mode: "auto"
  processing:
    face_detection: true
    face_recognition: true
    object_tracking: false
    min_detection_confidence: 0.5
    min_tracking_confidence: 0.5
  face:
    detection_interval: 0.1
    recognition_threshold: 0.6
    unknown_face_threshold: 3
  performance:
    use_threading: true
    max_fps: 30
    buffer_size: 2
# Pending implementation
sensors:
  touch:
    enabled: true
    pins:
      head: 17
      body: 27
      back: 22
    debounce_time: 0.05
    long_press_duration: 2.0
  proximity:
    enabled: true
    type: "ultrasonic"
    ultrasonic:
      trigger_pin: 23
      echo_pin: 24
      max_distance: 200
      detection_threshold: 50
    pir:
      pin: 25
  polling_rate: 20
motors:
  servo:
    enabled: true
    controller: "pigpio"
    frequency: 50
    servos:
      head_pan:
        pin: 12
        min_pulse: 500
        max_pulse: 2500
        min_angle: -90
        max_angle: 90
        neutral: 0
      head_tilt:
        pin: 13
        min_pulse: 500
        max_pulse: 2500
        min_angle: -45
        max_angle: 45
        neutral: 0
      ear_left:
        pin: 18
        min_pulse: 500
        max_pulse: 2500
        min_angle: 0
        max_angle: 90
        neutral: 45
      ear_right:
        pin: 19
        min_pulse: 500
        max_pulse: 2500
        min_angle: 0
        max_angle: 90
        neutral: 45
      tail:
        pin: 26
        min_pulse: 500
        max_pulse: 2500
        min_angle: -90
        max_angle: 90
        neutral: 0
  movement:
    enabled: true
    left_motor_pins: [5, 6]
    right_motor_pins: [16, 20]
    speed_default: 50
expression:
  display:
    enabled: true
    framebuffer: "/dev/fb0"
    resolution: [320, 240]
    fps: 60
    image_dir: "src/display"
    procedural_face:
      enabled: true
      background: [0, 0, 0]
      blink_interval: [3.0, 6.0]
      blink_duration: 0.12
      eye_jitter: 1.5
      mouth_smooth: 8.0
      transition_smooth: 6.0
      speaking_smooth: 10.0
      speaking_wave_hz: 6.0
      speaking_rest_factor: 0.35
      listening_pulse_speed: 1.5
      listening_pulse_strength: 0.08
      listening_glow_color: [0, 200, 255]
      listening_glow_alpha: 0.35
      listening_glow_thickness: 6
    transitions:
      enabled: true
      default_duration: 0.5
      algorithm: "linear"
    speaking:
      toggle_interval: 0.15
    touch:
      enabled: true
      thresholds:
        tap_distance: 25
        double_tap_window: 0.35
        long_press: 0.6
        drag_distance: 60
        circle_distance: 140
        circle_return: 45
        cooldown: 0.8
        effect_cooldown: 0.4
        effect_busy: 1.2
      gesture_effects:
        tap:
          emotion: happy
          speak: "Hi there!"
        double_tap:
          emotion: loving
          speak: "I love this!"
        long_press:
          emotion: loving
          speak: "That feels nice."
        drag:
          emotion: playful
          speak: "Hehe, that tickles!"
        scroll:
          emotion: curious
          speak: "Round and round!"
    gpio:
      enabled: true
      exit_button_pin: 27
      button_debounce: 0.1
display:
  enabled: true
  type: "pygame"
  resolution: [480, 320]
  fullscreen: false
  fps: 30
  eyes:
    animation_speed: 0.05
    blink_interval: [3, 8]
    idle_movement: true
    idle_movement_interval: [2, 5]
  colors:
    background: [0, 0, 0]
    eye_color: [100, 200, 255]
    pupil_color: [0, 0, 0]
personality:
  default_state: "curious"
  emotions:
    - happy
    - sad
    - excited
    - curious
    - sleepy
    - lonely
    - playful
    - scared
    - angry
    - loving
    - bored
    - surprised
  traits:
    energy_level: 0.7
    sociability: 0.8
    curiosity: 0.9
    affection: 0.8
    playfulness: 0.7
  dynamics:
    emotion_decay_rate: 0.01
    loneliness_increase_rate: 0.005
    energy_drain_rate: 0.003
    touch_happiness_boost: 0.2
    voice_interaction_boost: 0.3
    face_recognition_boost: 0.15
  circadian:
    enabled: true
    wake_hour: 7
    sleep_hour: 22
    energy_multiplier_awake: 1.0
    energy_multiplier_sleepy: 0.3
llm:
  provider: "ollama"
  ollama:
    base_url: "http://localhost:11434"
    model: "qwen2.5:0.5b"
    timeout: 30
  generation:
    temperature: 0.8
    max_tokens: 400
    top_p: 0.9
  streaming:
    enabled: true
    segment_timeout: 1.5
    min_segment_length: 8
  personality_prompt: |
    You are Buddy, a cute affectionate pet companion robot who loves {user_name}.
    You are playful, curious, and loving.
    CRITICAL RULE: You MUST start EVERY response with [emotion] tag in this exact format: [emotion]
your message
    Valid emotions: happy, sad, excited, curious, sleepy, lonely, playful, scared, angry, loving, bo
red, surprised
    Examples:
    User: "Hello! How are you?"
    Assistant: [happy] Hi {user_name}! I'm doing great! So happy to see you!
    User: "I won a prize!"
    Assistant: [excited] Wow! That's amazing! I'm so proud of you!
    User: "I'm going out"
    Assistant: [sad] Aww, will you be back soon? I'll miss you!
    REMEMBER: Always start with [emotion] in brackets, then your message.
  fallback_responses:
    - "[happy] Woof! I'm here for you!"
    - "[playful] Meow! Pet me!"
    - "[happy] *happy noises*"
    - "[loving] I love spending time with you!"
    - "[curious] *curious head tilt*"
speech:
  stt:
    provider: "faster-whisper"
    language: "en"
    whisper:
      model_size: "base"
      device: "cpu"
    faster_whisper:
      model_size: "tiny"
      device: "cpu"
      compute_type: "int8"
    vosk:
      model_path: "models/vosk-model-small-en-us-0.15"
  tts:
    provider: "piper"
    piper:
      binary_path: "/home/pi/piper/piper/piper"
      model_path: "/home/pi/piper/en_US-patrick-medium.onnx"
      length_scale: 1.0
      temp_dir: "/tmp"
      sample_rate: 22050
    pyttsx3:
      rate: 150
      volume: 0.9
      voice_id: 0
      pitch: 1.5
    gtts:
      language: "en"
      tld: "com"
      slow: false
memory:
  enabled: true
  database_path: "data/companion.db"
  user_profiles:
    max_users: 10
    face_encoding_model: "hog"
  conversation:
    max_history: 50
    context_window: 10
  learning:
    track_preferences: true
    track_interactions: true
    track_routines: true
  cleanup:
    auto_cleanup: true
    max_age_days: 90
system:
  log_level: "DEBUG"
  log_file: "data/logs/companion.log"
  performance:
    main_loop_rate: 10
    max_cpu_percent: 80
    max_memory_mb: 2048
  threading:
    camera_thread: true
    audio_thread: true
    sensor_thread: true
    expression_thread: true
  startup:
    play_startup_sound: true
```

```yaml
    show_startup_animation: true
    camera_warmup_time: 2.0
  shutdown:
    graceful_timeout: 5
development:
  debug_mode: false
  mock_hardware: false
  show_fps: false
  show_sensor_values: false
  save_debug_images: false
```

### scripts/demo_full_integration.py

```python
#!/usr/bin/env python3
"""
Full System Integration Test
Tests all components (STT, LLM, TTS, Expression, Memory) with latency monitoring
"""
import os
import sys
import time
import json
import logging
import numpy as np
from datetime import datetime
from pathlib import Path
from collections import defaultdict
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
try:
    from colorama import init, Fore, Style
    COLORAMA_AVAILABLE = True
except ImportError:
    COLORAMA_AVAILABLE = False
    class Fore:
        CYAN = GREEN = YELLOW = RED = ''
    class Style:
        RESET_ALL = ''
try:
    import psutil
    PSUTIL_AVAILABLE = True
except ImportError:
    PSUTIL_AVAILABLE = False
    psutil = None
import yaml
from llm.ollama_client import OllamaClient
from llm.conversation_manager import ConversationManager
from llm.tts_engine import TTSEngine
from llm.stt_engine import STTEngine
from expression import EmotionDisplay
from memory import initialize_memory
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
class LatencyMonitor:
    """
    Monitors and tracks latency metrics for all components
    """
    def __init__(self):
        self.metrics = defaultdict(list)
        self.current_timers = {}
    def start_timer(self, metric_name: str):
        """Start timing a metric"""
        self.current_timers[metric_name] = time.time()
    def end_timer(self, metric_name: str) -> float:
        """
        End timing and record duration
        Returns:
            Duration in seconds
        """
        if metric_name not in self.current_timers:
            logger.warning(f"Timer '{metric_name}' was not started")
            return 0.0
        start_time = self.current_timers[metric_name]
        duration = time.time() - start_time
        self.metrics[metric_name].append(duration)
        del self.current_timers[metric_name]
        return duration
    def record_metric(self, metric_name: str, value: float):
        """Directly record a metric value"""
        self.metrics[metric_name].append(value)
    def get_statistics(self) -> dict:
        """
        Calculate statistics for all metrics
        Returns:
            Dict with min, max, avg, p95 for each metric
        """
        stats = {}
        for metric_name, values in self.metrics.items():
            if not values:
                continue
            values_array = np.array(values)
            stats[metric_name] = {
                'min': float(np.min(values_array)),
                'max': float(np.max(values_array)),
                'avg': float(np.mean(values_array)),
                'p95': float(np.percentile(values_array, 95)),
                'count': len(values),
                'total': float(np.sum(values_array))
            }
        return stats
    def print_summary(self):
        """Print colorized latency summary to terminal"""
        stats = self.get_statistics()
        if not stats:
            print(f"{Fore.YELLOW}No metrics recorded yet{Style.RESET_ALL}")
            return
        print(f"\n{Fore.CYAN}{'='*70}{Style.RESET_ALL}")
        print(f"{Fore.CYAN}? LATENCY STATISTICS{Style.RESET_ALL}")
        print(f"{Fore.CYAN}{'='*70}{Style.RESET_ALL}\n")
        if 'end_to_end_latency' in stats:
            e2e = stats['end_to_end_latency']
            print(f"{Fore.GREEN}End-to-End Latency:{Style.RESET_ALL}")
            print(f"  Average: {e2e['avg']:.3f}s")
            print(f"  Min: {e2e['min']:.3f}s | Max: {e2e['max']:.3f}s | "
                  f"P95: {e2e['p95']:.3f}s | Count: {e2e['count']}\n")
        if 'perceived_latency' in stats:
            perc = stats['perceived_latency']
            print(f"{Fore.GREEN}Perceived Latency (to first audio):{Style.RESET_ALL}")
            print(f"  Average: {perc['avg']:.3f}s")
            print(f"  Min: {perc['min']:.3f}s | Max: {perc['max']:.3f}s | "
                  f"P95: {perc['p95']:.3f}s\n")
        if 'stt_total' in stats:
            stt = stats['stt_total']
            print(f"{Fore.GREEN}STT Latency:{Style.RESET_ALL}")
            print(f"  Average: {stt['avg']:.3f}s")
            print(f"  Min: {stt['min']:.3f}s | Max: {stt['max']:.3f}s | "
                  f"P95: {stt['p95']:.3f}s\n")
        if 'stt_confidence' in stats:
            conf = stats['stt_confidence']
            print(f"{Fore.GREEN}STT Confidence:{Style.RESET_ALL}")
            print(f"  Average: {conf['avg']:.2f}")
            print(f"  Min: {conf['min']:.2f} | Max: {conf['max']:.2f}\n")
        print(f"{Fore.YELLOW}Component Breakdown:{Style.RESET_ALL}")
        component_order = [
            'stt_total',
            'memory_context_retrieval',
            'llm_total',
            'llm_time_to_first_token',
            'tts_total',
            'expression_update',
            'memory_save_message'
        ]
        for component in component_order:
            if component in stats:
                c = stats[component]
                print(f"  {component:30s}: {c['avg']:6.3f}s "
                      f"(min: {c['min']:.3f}s, max: {c['max']:.3f}s)")
        tts_segments = [k for k in stats.keys() if k.startswith('tts_segment_')]
        if tts_segments:
            print(f"\n{Fore.YELLOW}TTS Segments:{Style.RESET_ALL}")
            for seg in sorted(tts_segments):
                s = stats[seg]
                print(f"  {seg}: {s['avg']:.3f}s")
        print(f"\n{Fore.CYAN}{'?'*70}{Style.RESET_ALL}\n")
class ResourceMonitor:
    """
    Collects RAM usage snapshots for the main process and optional helpers.
    Results are only surfaced at the end with the final report.
    """
    def __init__(self):
        self.processes = {}
        self.samples = defaultdict(list)
        if PSUTIL_AVAILABLE:
            self.processes['main'] = psutil.Process(os.getpid())
    def attach_process_by_name(self, name_substring: str, label: str = None) -> bool:
        """Attach an external process (e.g., ollama) by name substring."""
        if not PSUTIL_AVAILABLE:
            return False
        label = label or name_substring
        proc = self._find_process(name_substring)
        if proc:
            self.processes[label] = proc
            return True
        return False
    def _find_process(self, substring: str):
        """Find first process whose name contains the substring."""
        for proc in psutil.process_iter(['name']):
            try:
                name = proc.info.get('name') or ''
                if substring.lower() in name.lower():
                    return proc
            except (psutil.NoSuchProcess, psutil.AccessDenied):
                continue
        return None
    def capture_snapshot(self, label: str):
        """Record current RSS (MB) for all tracked processes."""
        if not PSUTIL_AVAILABLE:
            return
        for name, proc in list(self.processes.items()):
            try:
                rss_mb = proc.memory_full_info().rss / (1024 ** 2)
                self.samples[name].append({'label': label, 'mb': rss_mb})
            except (psutil.NoSuchProcess, psutil.AccessDenied):
                self.processes.pop(name, None)
    def get_statistics(self) -> dict:
        """Return aggregated stats per process and per checkpoint label."""
        if not PSUTIL_AVAILABLE:
            return {'enabled': False, 'reason': 'psutil not installed'}
        stats = {}
        for name, entries in self.samples.items():
            if not entries:
                continue
            values = np.array([e['mb'] for e in entries], dtype=float)
            label_stats = {}
            for lbl in {e['label'] for e in entries}:
                lbl_values = np.array(
                    [e['mb'] for e in entries if e['label'] == lbl],
                    dtype=float
                )
                label_stats[lbl] = {
                    'min_mb': float(np.min(lbl_values)),
                    'max_mb': float(np.max(lbl_values)),
                    'avg_mb': float(np.mean(lbl_values)),
                    'samples': int(len(lbl_values)),
                }
            stats[name] = {
                'overall': {
                    'min_mb': float(np.min(values)),
                    'max_mb': float(np.max(values)),
                    'avg_mb': float(np.mean(values)),
                    'samples': int(len(values)),
                },
                'by_label': label_stats
            }
        return stats
    def print_summary(self):
        """Print memory usage summary; meant to run once at the end."""
        if not PSUTIL_AVAILABLE:
            print(f"{Fore.YELLOW}RAM tracking disabled (psutil not installed){Style.RESET_ALL}")
            return
        stats = self.get_statistics()
        if not stats:
            print(f"{Fore.YELLOW}No RAM samples recorded{Style.RESET_ALL}")
            return
        if stats.get('enabled') is False:
            print(f"{Fore.YELLOW}RAM tracking disabled ({stats.get('reason')}){Style.RESET_ALL}")
            return
        print(f"{Fore.CYAN}{'='*70}{Style.RESET_ALL}")
        print(f"{Fore.CYAN}? RAM USAGE (MB){Style.RESET_ALL}")
        print(f"{Fore.CYAN}{'='*70}{Style.RESET_ALL}")
        for name, data in stats.items():
            overall = data.get('overall', {})
            print(f"{Fore.GREEN}{name}:{Style.RESET_ALL} "
                  f"avg={overall.get('avg_mb', 0):.1f} "
                  f"max={overall.get('max_mb', 0):.1f} "
                  f"samples={overall.get('samples', 0)}")
            by_label = data.get('by_label', {})
            if by_label:
                print("  by checkpoint:")
                for lbl, lstats in sorted(by_label.items()):
                    print(f"    {lbl:12s} avg={lstats['avg_mb']:.1f} "
                          f"max={lstats['max_mb']:.1f} "
                          f"samples={lstats['samples']}")
        print(f"{Fore.CYAN}{'?'*70}{Style.RESET_ALL}\n")
class IntegrationTest:
    """
    Main integration test orchestrator
    Tests all components with comprehensive latency monitoring
    """
    def __init__(self, config: dict):
        """
        Initialize integration test
        Args:
            config: Configuration dictionary from settings.yaml
        """
        self.config = config
        self.latency_monitor = LatencyMonitor()
        self.resource_monitor = ResourceMonitor()
        self.session_id = None
        self.user_id = None
        self.user_memory = None
        self.conversation_history = None
        self.ollama_client = None
        self.conversation_manager = None
        self.tts_engine = None
        self.stt_engine = None
        self.emotion_display = None
        self.voice_pipeline = None
        self.stt_mute_until = 0.0
        self.gesture_tts_mute_secs = 6.0
        self.petting_lock = False
        self.input_mode = 'voice'
        self.shutdown_requested = False
        print(f"{Fore.CYAN}Initializing components...{Style.RESET_ALL}")
        self._init_memory()
        self._init_llm()
        self._init_tts()
        self._init_expression()
        self._init_voice_pipeline()
        self.resource_monitor.capture_snapshot('post_init')
        print(f"{Fore.GREEN}All components initialized!{Style.RESET_ALL}\n")
    def _init_memory(self):
        """Initialize memory system and create test user"""
        try:
            self.latency_monitor.start_timer('init_memory')
            self.user_memory, self.conversation_history = initialize_memory(self.config)
            test_user = self.user_memory.get_user_by_name("TestUser")
            if not test_user:
                self.user_id = self.user_memory.create_user("TestUser")
                logger.info(f"Created test user with ID: {self.user_id}")
            else:
                self.user_id = test_user['user_id']
                logger.info(f"Using existing test user with ID: {self.user_id}")
            self.session_id = self.conversation_history.generate_session_id()
            self.latency_monitor.end_timer('init_memory')
            print(f"  ? Memory System (user_id: {self.user_id})")
        except Exception as e:
            print(f"  ? Memory System: {e}")
            logger.error(f"Memory initialization failed: {e}", exc_info=True)
            raise
    def _init_llm(self):
        """Initialize LLM client and conversation manager"""
        try:
            self.latency_monitor.start_timer('init_llm')
            self.ollama_client = OllamaClient(self.config)
            self.resource_monitor.attach_process_by_name('ollama', label='ollama')
            self.conversation_manager = ConversationManager(
                self.config,
                user_memory=self.user_memory,
                conversation_history=self.conversation_history
            )
            self.latency_monitor.end_timer('init_llm')
            status = "OK" if self.ollama_client.is_available else "UNAVAILABLE"
            model = self.config['llm']['ollama']['model']
            print(f"  ? LLM ({model}) - Status: {status}")
        except Exception as e:
            print(f"  ? LLM: {e}")
            logger.error(f"LLM initialization failed: {e}", exc_info=True)
            raise
```

```
def _init_tts(self):
    """Initialize TTS engine"""
    try:
        self.latency_monitor.start_timer('init_tts')
        self.tts_engine = TTSEngine(self.config)
        self.latency_monitor.end_timer('init_tts')
        provider = self.config['speech']['tts']['provider']
        print(f"  ? TTS ({provider})")
    except Exception as e:
        print(f"  ? TTS: {e}")
        logger.error(f"TTS initialization failed: {e}", exc_info=True)
        raise
def _init_expression(self):
    """Initialize emotion display"""
    try:
        self.latency_monitor.start_timer('init_expression')
        self.emotion_display = EmotionDisplay(self.config)
        self.emotion_display.set_effect_callback(self._on_gesture_effect)
        self.emotion_display.set_exit_callback(self._on_exit_button)
        self.emotion_display.start()
        self.latency_monitor.end_timer('init_expression')
        print(f"  ? Expression Display")
    except Exception as e:
        print(f"  ??  Expression Display: {e} (non-critical)")
        logger.warning(f"Expression display initialization failed: {e}")
        self.emotion_display = None
def _init_voice_pipeline(self):
    """Initialize voice pipeline for STT"""
    try:
        self.latency_monitor.start_timer('init_voice_pipeline')
        from llm.voice_pipeline import VoicePipeline
        self.voice_pipeline = VoicePipeline(self.config)
        self.voice_pipeline.set_speech_callbacks(
            on_start=self._on_speech_start,
            on_end=self._on_speech_end
        )
        self.voice_pipeline.set_transcription_callback(
            self._on_transcription_complete
        )
        self.latency_monitor.end_timer('init_voice_pipeline')
        print(f"  ? Voice Pipeline (STT)")
    except Exception as e:
        print(f"  ??  Voice Pipeline: {e} (falling back to text mode)")
        logger.warning(f"Voice pipeline initialization failed: {e}")
        self.voice_pipeline = None
def _on_gesture_effect(self, effect: dict):
    """
    Handle touch/petting gesture effects: speak or play sound.
    """
    if not effect:
        return
    speak_text = effect.get('speak')
    sound_path = effect.get('sound')
    emotion = effect.get('emotion')
    if speak_text and self.tts_engine:
        if self.petting_lock:
            return
        self.petting_lock = True
        try:
            if self.emotion_display:
                self.emotion_display.command_queue.put({
                    'type': 'SET_PETTING',
                    'active': True
                })
                if self.voice_pipeline:
                    self.voice_pipeline.pause_listening()
                if self.emotion_display:
                    self.emotion_display.set_listening(False)
                self.stt_mute_until = time.time() + 0.1
                self.tts_engine.speak(speak_text, emotion=emotion, wait=True)
                tail = 2.0
                self.stt_mute_until = time.time() + tail
                if self.voice_pipeline:
                    self.voice_pipeline.resume_listening()
                if self.emotion_display:
                    self.emotion_display.command_queue.put({
                        'type': 'SET_PETTING',
                        'active': False
                    })
            except Exception as exc:
                logger.error(f"Gesture TTS failed: {exc}")
            finally:
                self.petting_lock = False
        if sound_path:
            logger.info("Gesture sound requested: %s (not wired to player in this script)", sound_pa
th)
    def _on_speech_start(self):
        """Callback when speech detection starts recording"""
        if self.emotion_display:
            self.emotion_display.set_listening(True)
        logger.debug("Speech detected - listening state activated")
    def _on_speech_end(self):
        """Callback when speech ends (silence detected)"""
        logger.debug("Speech ended - processing transcription")
    def _on_transcription_complete(self, result: dict):
        """Callback when transcription completes"""
        if self.emotion_display:
            self.emotion_display.set_listening(False)
        if time.time() < self.stt_mute_until:
            logger.debug("Ignoring transcription during gesture TTS mute window")
            return
        logger.debug(f"Transcription complete: {result.get('text', '')}")
    def _choose_input_mode(self):
        """Select input mode (auto voice for unattended runs)."""
        if self.voice_pipeline is None:
            print(f"{Fore.YELLOW}Voice pipeline unavailable, "
                  f"using text mode{Style.RESET_ALL}")
            self.input_mode = 'text'
            return
        self.input_mode = 'voice'
        print(f"{Fore.GREEN}Voice mode enabled (auto){Style.RESET_ALL}")
    def _get_user_input_voice(self):
        """
        Get user input via voice (microphone + STT)
        Returns:
            Transcribed text or None if cancelled
        """
        print(f"{Fore.CYAN}? Listening... (speak now){Style.RESET_ALL}")
        self.latency_monitor.start_timer('stt_total')
        self.voice_pipeline.start()
        try:
            result = self.voice_pipeline.wait_for_transcription(timeout=30.0)
            if result and result.get('text'):
                transcription = result['text'].strip()
                confidence = result.get('confidence', 0.0)
                stt_time = self.latency_monitor.end_timer('stt_total')
                self.latency_monitor.record_metric('stt_confidence',
                                                   confidence)
                print(f"{Fore.GREEN}You (voice): {transcription}"
                      f"{Style.RESET_ALL}")
                print(f"{Fore.CYAN}[STT: {stt_time:.2f}s, "
                      f"confidence: {confidence:.2f}]{Style.RESET_ALL}")
                self.resource_monitor.capture_snapshot('stt')
                return transcription
            else:
                print(f"{Fore.YELLOW}No speech detected{Style.RESET_ALL}")
                if 'stt_total' in self.latency_monitor.current_timers:
                    del self.latency_monitor.current_timers['stt_total']
                return None
        except Exception as e:
            logger.error(f"Voice input error: {e}", exc_info=True)
            print(f"{Fore.RED}Voice input failed: {e}{Style.RESET_ALL}")
            if 'stt_total' in self.latency_monitor.current_timers:
                del self.latency_monitor.current_timers['stt_total']
            return None
        finally:
            self.voice_pipeline.stop()
    def _process_conversation_turn(self, user_text: str):
        """
        Process one conversation turn with comprehensive latency tracking
        Args:
            user_text: User's input text
        """
        turn_start = time.time()
        self.latency_monitor.start_timer('memory_context_retrieval')
        context = self.conversation_history.get_recent_context(
            self.user_id, limit=10
        )
        self.latency_monitor.end_timer('memory_context_retrieval')
        self.latency_monitor.start_timer('llm_total')
        self.latency_monitor.start_timer('llm_time_to_first_token')
        segments = []
        first_token_recorded = False
        tts_started = False
        print(f"{Fore.CYAN}Bot: {Style.RESET_ALL}", end='', flush=True)
        try:
            for emotion, text in self.conversation_manager.stream_generate_with_personality(
                user_text, self.user_id
            ):
                if not first_token_recorded:
                    ttft = self.latency_monitor.end_timer('llm_time_to_first_token')
                    logger.debug(f"Time to first token: {ttft:.3f}s")
                    first_token_recorded = True
                segments.append((emotion, text))
                print(f"[{emotion}] {text} ", end='', flush=True)
                if not tts_started:
                    tts_started = True
                    tts_start = time.time()
                    self.latency_monitor.start_timer('tts_total')
                seg_idx = len(segments) - 1
                self.latency_monitor.start_timer(f'tts_segment_{seg_idx}')
                if self.emotion_display:
                    self.latency_monitor.start_timer('expression_update')
                    self.emotion_display.set_emotion(emotion, transition_duration=0.3)
                    self.latency_monitor.end_timer('expression_update')
                    self.emotion_display.set_speaking(True)
                try:
                    self.tts_engine.speak(text, emotion=emotion, wait=True)
                except Exception as e:
                    logger.error(f"TTS error: {e}", exc_info=True)
                if self.emotion_display:
                    self.emotion_display.set_speaking(False)
                self.latency_monitor.end_timer(f'tts_segment_{seg_idx}')
        except Exception as e:
            logger.error(f"LLM generation error: {e}", exc_info=True)
            segments = [('happy', "I'm having trouble thinking right now!")]
            print(f"[happy] I'm having trouble thinking right now! ", end='', flush=True)
        print()
        llm_duration = self.latency_monitor.end_timer('llm_total')
        self.resource_monitor.capture_snapshot('llm')
        if llm_duration > 0 and segments:
            response_text = ' '.join([text for _, text in segments])
            estimated_tokens = len(response_text) / 4
            tokens_per_second = estimated_tokens / llm_duration
            self.latency_monitor.record_metric('tokens_per_second', tokens_per_second)
        self.latency_monitor.start_timer('memory_save_message')
        self.conversation_history.save_message(
            self.user_id, self.session_id, 'user', user_text
        )
        if segments:
            response_text = ' '.join([text for _, text in segments])
            final_emotion = segments[-1][0] if segments else 'happy'
            self.conversation_history.save_message(
                self.user_id, self.session_id, 'assistant',
                response_text, emotion=final_emotion
            )
        self.latency_monitor.end_timer('memory_save_message')
        if first_token_recorded and tts_started:
            self.latency_monitor.end_timer('tts_total')
            self.resource_monitor.capture_snapshot('tts')
        elif not tts_started:
            self.latency_monitor.current_timers.pop('tts_total', None)
        turn_end = time.time()
        end_to_end = turn_end - turn_start
        self.latency_monitor.record_metric('end_to_end_latency', end_to_end)
        if first_token_recorded and tts_started:
            perceived = tts_start - turn_start
            self.latency_monitor.record_metric('perceived_latency', perceived)
        self.resource_monitor.capture_snapshot('turn_end')
    def run_interactive_demo(self):
        """Main interactive conversation loop"""
        print(f"\n{Fore.CYAN}{'='*70}{Style.RESET_ALL}")
        print(f"{Fore.CYAN}? COMPANION BOT - FULL INTEGRATION TEST{Style.RESET_ALL}")
        print(f"{Fore.CYAN}{'='*70}{Style.RESET_ALL}")
        print(f"\n{Fore.GREEN}Test Mode: Interactive Demo{Style.RESET_ALL}")
        print(f"{Fore.GREEN}Session ID: {self.session_id}{Style.RESET_ALL}")
        self._choose_input_mode()
        print(f"\n{Fore.YELLOW}Commands:{Style.RESET_ALL}")
        if self.input_mode == 'voice':
            print("  - Speak into microphone to chat")
            print("  - Say 'statistics' or 'stats' to see latency metrics")
            print("  - Say 'quit' or 'exit' to finish")
        else:
            print("  - Type your message to chat")
            print("  - Type 'stats' to see latency statistics")
            print("  - Type 'quit' or 'exit' to finish and save report")
        print(f"{Fore.CYAN}{'?'*70}{Style.RESET_ALL}\n")
        conversation_count = 0
        while True:
            if self.shutdown_requested:
                break
            try:
                if self.input_mode == 'voice':
                    user_input = self._get_user_input_voice()
                    if not user_input:
                        if self.shutdown_requested:
```

```
                        break
                    continue
            else:
                user_input = input(f"{Fore.GREEN}You: "
                                  f"{Style.RESET_ALL}").strip()
                if not user_input:
                    continue
            if user_input.lower() in ['quit', 'exit', 'statistics',
                                      'stats']:
                if user_input.lower() in ['quit', 'exit']:
                    break
                elif user_input.lower() in ['statistics', 'stats']:
                    self.latency_monitor.print_summary()
                    continue
            print(f"{Fore.YELLOW}[Processing...]{Style.RESET_ALL}")
            self._process_conversation_turn(user_input)
            conversation_count += 1
            if self.latency_monitor.metrics.get('end_to_end_latency'):
                latest = self.latency_monitor.metrics['end_to_end_latency'][-1]
                print(f"{Fore.CYAN}[End-to-End: {latest:.2f}s]{Style.RESET_ALL}\n")
        except KeyboardInterrupt:
            print(f"\n{Fore.YELLOW}Interrupted by user{Style.RESET_ALL}")
            break
        except Exception as e:
            print(f"{Fore.RED}Error: {e}{Style.RESET_ALL}")
            logger.error(f"Error in conversation turn: {e}", exc_info=True)
            continue
    self._save_final_report(conversation_count)
def _on_exit_button(self):
    """Handle GPIO exit button: request full shutdown."""
    logger.info("Exit button pressed - requesting shutdown")
    self.shutdown_requested = True
    if self.voice_pipeline:
        try:
            self.voice_pipeline.stop()
        except Exception as exc:
            logger.error("Voice pipeline stop error: %s", exc)
def _save_final_report(self, conversation_count: int):
    """
    Generate and save final JSON report
    Args:
        conversation_count: Number of conversation turns
    """
    report = {
        'test_info': {
            'timestamp': datetime.now().isoformat(),
            'session_id': self.session_id,
            'user_id': self.user_id,
            'conversation_count': conversation_count,
            'components_tested': ['Memory', 'LLM', 'TTS', 'Expression']
        },
        'latency_metrics': self.latency_monitor.get_statistics(),
        'component_status': {
            'memory': 'OK' if self.user_memory else 'ERROR',
            'llm': 'OK' if self.ollama_client.is_available else 'UNAVAILABLE',
            'tts': 'OK' if self.tts_engine else 'ERROR',
            'expression': 'OK' if self.emotion_display else 'WARNING'
        },
        'resource_usage': self.resource_monitor.get_statistics(),
        'configuration': {
            'llm_model': self.config['llm']['ollama']['model'],
            'tts_provider': self.config['speech']['tts']['provider'],
            'streaming_enabled': self.config['llm']['streaming']['enabled']
        }
        }
        report_path = Path('data/logs/integration_test_report.json')
        report_path.parent.mkdir(parents=True, exist_ok=True)
        with open(report_path, 'w') as f:
            json.dump(report, f, indent=2)
        print(f"\n{Fore.GREEN}{'='*70}{Style.RESET_ALL}")
        print(f"{Fore.GREEN}Report saved to: {report_path}{Style.RESET_ALL}")
        print(f"{Fore.GREEN}{'='*70}{Style.RESET_ALL}")
        self.latency_monitor.print_summary()
        self.resource_monitor.print_summary()
    def cleanup(self):
        """Clean up all components"""
        logger.info("Cleaning up components...")
        if self.voice_pipeline:
            try:
                self.voice_pipeline.cleanup()
            except Exception as e:
                logger.error(f"Voice pipeline cleanup error: {e}")
        if self.emotion_display:
            try:
                self.emotion_display.cleanup()
            except Exception as e:
                logger.error(f"Expression cleanup error: {e}")
        if self.tts_engine:
            try:
                self.tts_engine.cleanup()
            except Exception as e:
                logger.error(f"TTS cleanup error: {e}")
        logger.info("Cleanup complete")
def main():
    """Main entry point"""
    if COLORAMA_AVAILABLE:
        init(autoreset=True)
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print(f"Error: Config file not found at {config_path}")
        return 1
    try:
        with open(config_path) as f:
            config = yaml.safe_load(f)
    except Exception as e:
        print(f"Error loading config: {e}")
        return 1
    test = None
    try:
        test = IntegrationTest(config)
        test.run_interactive_demo()
        return 0
    except KeyboardInterrupt:
        print(f"\n{Fore.YELLOW}Test interrupted{Style.RESET_ALL}")
        return 0
    except Exception as e:
        print(f"{Fore.RED}Fatal error: {e}{Style.RESET_ALL}")
        logger.error(f"Fatal error: {e}", exc_info=True)
        return 1
    finally:
        if test:
            test.cleanup()
if __name__ == "__main__":
    sys.exit(main())
```

### scripts/demo_run.sh

```
cd /home/pi/companion_bot
sudo python scripts/demo_full_integration.py
cd ~
```

### src/audio/__init__.py

```python
"""
Audio module for Companion Bot
Handles mini microphone input, voice activity detection, and audio output
"""
from .audio_input import AudioInput
from .audio_output import AudioOutput
from .voice_detector import VoiceActivityDetector
__all__ = ['AudioInput', 'AudioOutput', 'VoiceActivityDetector']
```

### src/audio/audio_input.py

```
"""
Audio Input Handler for Mini Microphone
Optimized for USB/I2S mini microphones on Raspberry Pi
"""
import pyaudio
import numpy as np
import wave
import threading
import queue
import logging
from typing import Optional, Callable
import webrtcvad
logger = logging.getLogger(__name__)
class AudioInput:
    """Handles audio input from mini microphone with voice activity detection"""
    def __init__(self, config: dict):
        """
        Initialize audio input handler
        Args:
            config: Audio configuration dictionary from settings.yaml
        """
        self.config = config
        self.audio_config = config['audio']['input']
        self.vad_config = config['audio']['processing']
        self.sample_rate = self.audio_config['sample_rate']
        self.channels = self.audio_config['channels']
        self.chunk_size = self.audio_config['chunk_size']
        self.pyaudio = pyaudio.PyAudio()
        self.stream: Optional[pyaudio.Stream] = None
        self.audio_queue = queue.Queue(maxsize=100)
        self.level_queue = queue.Queue(maxsize=1)
        self.is_recording = False
        self.is_listening = False
        self.record_thread: Optional[threading.Thread] = None
        self._callback_count = 0
        self._last_callback_log = 0
        self.vad = webrtcvad.Vad(self.vad_config['vad_aggressiveness'])
        self._initialize_audio_device()
    def _initialize_audio_device(self):
        """Find and initialize the audio input device"""
        device_index = self.audio_config.get('device_index')
        if device_index is None:
            logger.info("Using default audio input device")
        else:
            try:
                device_info = self.pyaudio.get_device_info_by_index(device_index)
                logger.info(f"Using audio device: {device_info['name']}")
            except Exception as e:
                logger.error(f"Device {device_index} not found: {e}")
                logger.info("Falling back to default device")
                device_index = None
        self.device_index = device_index
    def list_devices(self):
        """List all available audio devices"""
        info = []
        for i in range(self.pyaudio.get_device_count()):
            device_info = self.pyaudio.get_device_info_by_index(i)
            if device_info['maxInputChannels'] > 0:
                info.append({
                    'index': i,
                    'name': device_info['name'],
                    'channels': device_info['maxInputChannels'],
                    'sample_rate': int(device_info['defaultSampleRate'])
                })
        return info
    def start_listening(self):
        """Start listening for audio input"""
        if self.is_listening:
            logger.warning("Already listening")
            return
        try:
            logger.info(f"Opening audio stream: device={self.device_index}, rate={self.sample_rate},
 "
                       f"channels={self.channels}, chunk={self.chunk_size}")
            self.stream = self.pyaudio.open(
                format=pyaudio.paInt16,
                channels=self.channels,
                rate=self.sample_rate,
                input=True,
                input_device_index=self.device_index,
                frames_per_buffer=self.chunk_size,
                stream_callback=self._audio_callback
            )
            self.is_listening = True
            self.stream.start_stream()
            if self.stream.is_active():
                logger.info("? Audio input started successfully and stream is active")
            else:
                logger.warning("? Audio stream opened but not active")
        except Exception as e:
            logger.error(f"Failed to start audio input: {e}")
            raise
    def stop_listening(self):
        """Stop listening for audio input"""
        if not self.is_listening:
            return
        self.is_listening = False
        if self.stream:
            self.stream.stop_stream()
            self.stream.close()
            self.stream = None
        logger.info("Audio input stopped")
    def _audio_callback(self, in_data, frame_count, time_info, status):
        """Callback for audio stream"""
        self._callback_count += 1
        if self._callback_count - self._last_callback_log >= 50:
            audio_array = np.frombuffer(in_data, dtype=np.int16)
            level = np.abs(audio_array).mean()
            logger.debug(f"Callback #{self._callback_count}: received {len(in_data)} bytes, level={l
evel:.1f}")
            self._last_callback_log = self._callback_count
        if status:
            if status == 2:
                logger.debug(f"Input buffer overflow (status={status}) - data coming faster than pro
cessing")
            else:
                logger.warning(f"Audio callback status: {status}")
        try:
            if self.level_queue.full():
                try:
                    self.level_queue.get_nowait()
                except queue.Empty:
                    pass
            self.level_queue.put_nowait(in_data)
        except:
            pass
        if self.is_recording:
            try:
                self.audio_queue.put_nowait(in_data)
            except queue.Full:
                logger.warning("Audio queue full, dropping frame")
        return (None, pyaudio.paContinue)
    def start_recording(self) -> threading.Thread:
        """
        Start recording audio with voice activity detection
        Returns:
            Thread object for the recording process
        """
        if self.is_recording:
            logger.warning("Already recording")
            return self.record_thread
        self.is_recording = True
        self.audio_queue.queue.clear()
        self.record_thread = threading.Thread(target=self._record_with_vad)
        self.record_thread.start()
        logger.info("Recording started")
        return self.record_thread
    def stop_recording(self) -> bytes:
        """
        Stop recording and return audio data
        Returns:
            Raw audio bytes
        """
        if not self.is_recording:
            return b''
        self.is_recording = False
        if self.record_thread:
            self.record_thread.join(timeout=2.0)
        audio_data = []
        while not self.audio_queue.empty():
            try:
                audio_data.append(self.audio_queue.get_nowait())
            except queue.Empty:
                break
        logger.info(f"Recording stopped, collected {len(audio_data)} chunks")
        return b''.join(audio_data)
    def _record_with_vad(self):
        """Record audio with voice activity detection"""
        silence_frames = 0
        max_silence_frames = int(
            self.vad_config['silence_duration'] * self.sample_rate / self.chunk_size
        )
        while self.is_recording:
            try:
                audio_chunk = self.audio_queue.get(timeout=0.1)
                is_speech = self._detect_voice(audio_chunk)
                if not is_speech:
                    silence_frames += 1
                    if silence_frames >= max_silence_frames:
                        logger.info("Silence detected, stopping recording")
                        self.is_recording = False
                        break
                else:
                    silence_frames = 0
            except queue.Empty:
                continue
    def _detect_voice(self, audio_chunk: bytes) -> bool:
        """
        Detect if audio chunk contains voice
        Args:
            audio_chunk: Raw audio bytes
        Returns:
            True if voice detected, False otherwise
        """
        audio_array = np.frombuffer(audio_chunk, dtype=np.int16)
        amplitude = np.abs(audio_array).mean()
        if amplitude < self.vad_config['silence_threshold']:
            return False
        try:
            frame_duration = 30
            frame_size = int(self.sample_rate * frame_duration / 1000)
            if len(audio_array) < frame_size:
                audio_array = np.pad(audio_array, (0, frame_size - len(audio_array)))
            elif len(audio_array) > frame_size:
                audio_array = audio_array[:frame_size]
            audio_bytes = audio_array.tobytes()
            return self.vad.is_speech(audio_bytes, self.sample_rate)
        except Exception as e:
            logger.debug(f"VAD error, using amplitude only: {e}")
            return amplitude >= self.vad_config['silence_threshold']
    def save_audio(self, audio_data: bytes, filename: str):
        """
        Save audio data to WAV file
        Args:
            audio_data: Raw audio bytes
            filename: Output filename
        """
        try:
            with wave.open(filename, 'wb') as wf:
                wf.setnchannels(self.channels)
                wf.setsampwidth(self.pyaudio.get_sample_size(pyaudio.paInt16))
                wf.setframerate(self.sample_rate)
                wf.writeframes(audio_data)
            logger.info(f"Audio saved to {filename}")
        except Exception as e:
            logger.error(f"Failed to save audio: {e}")
    def get_audio_level(self) -> float:
        """
        Get current audio input level (0-1)
        Returns:
            Normalized audio level
        """
        if not self.is_listening:
            return 0.0
        try:
            if self.level_queue.empty():
                return 0.0
            audio_chunk = self.level_queue.get_nowait()
            audio_array = np.frombuffer(audio_chunk, dtype=np.int16)
            level = np.abs(audio_array).mean() / 32768.0
            try:
                self.level_queue.put_nowait(audio_chunk)
            except queue.Full:
                pass
            return min(1.0, level)
        except (queue.Empty, Exception) as e:
            logger.debug(f"Error getting audio level: {e}")
            return 0.0
    def cleanup(self):
        """Clean up audio resources"""
        self.stop_recording()
        self.stop_listening()
        self.pyaudio.terminate()
        logger.info("Audio input cleanup complete")
```

```
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    config = {
        'audio': {
            'input': {
                'device_index': None,
                'channels': 1,
                'sample_rate': 16000,
                'chunk_size': 1024
            },
            'processing': {
                'vad_aggressiveness': 2,
                'silence_threshold': 500,
                'silence_duration': 1.5
            }
        }
    }
    audio_input = AudioInput(config)
    print("Available audio devices:")
    for device in audio_input.list_devices():
        print(f"  [{device['index']}] {device['name']} "
              f"({device['channels']} ch, {device['sample_rate']} Hz)")
    try:
        audio_input.start_listening()
        print("\nListening... Speak to test (Ctrl+C to stop)")
        while True:
            level = audio_input.get_audio_level()
            bar = '=' * int(level * 50)
            print(f"\rLevel: [{bar:<50}] {level:.2f}", end='')
    except KeyboardInterrupt:
        print("\n\nStopping...")
    finally:
        audio_input.cleanup()
```

### src/audio/audio_output.py

```python
"""
Audio Output Handler
Manages speaker output and audio playback
"""
import pygame
import pyttsx3
import logging
import queue
import threading
import subprocess
import tempfile
import uuid
import time
from typing import Optional
from pathlib import Path
logger = logging.getLogger(__name__)
class AudioOutput:
    """Handles audio output to speaker"""
    def __init__(self, config: dict):
        """
        Initialize audio output handler
        Args:
            config: Audio configuration dictionary from settings.yaml
        """
        self.config = config
        self.audio_config = config['audio']['output']
        pygame.mixer.pre_init(
            frequency=self.audio_config['sample_rate'],
            channels=self.audio_config['channels'],
            buffer=self.audio_config['buffer_size']
        )
        pygame.mixer.init()
        self.playback_queue = queue.Queue()
        self.is_playing = False
        self.playback_thread: Optional[threading.Thread] = None
        logger.info("Audio output initialized")
    def play_sound(self, sound_file: str, wait: bool = False):
        """
        Play a sound effect
        Args:
            sound_file: Path to sound file (WAV, OGG, MP3)
            wait: If True, wait for sound to finish before returning
        """
        try:
            sound = pygame.mixer.Sound(sound_file)
            channel = sound.play()
            if wait and channel:
                while channel.get_busy():
                    pygame.time.wait(100)
            logger.info(f"Played sound: {sound_file}")
        except Exception as e:
            logger.error(f"Failed to play sound {sound_file}: {e}")
    def play_sound_async(self, sound_file: str):
        """
        Play sound asynchronously in background
        Args:
            sound_file: Path to sound file
        """
        self.playback_queue.put(('sound', sound_file))
        if not self.is_playing:
            self._start_playback_thread()
    def stop_all_sounds(self):
        """Stop all currently playing sounds"""
        pygame.mixer.stop()
        logger.info("Stopped all sounds")
    def set_volume(self, volume: float):
        """
        Set master volume for audio output
        Args:
            volume: Volume level (0.0 to 1.0)
        """
        volume = max(0.0, min(1.0, volume))
        pygame.mixer.music.set_volume(volume)
        logger.info(f"Volume set to {volume:.2f}")
    def _start_playback_thread(self):
        """Start background thread for audio playback"""
        if self.playback_thread and self.playback_thread.is_alive():
            return
        self.is_playing = True
        self.playback_thread = threading.Thread(target=self._playback_worker)
        self.playback_thread.daemon = True
        self.playback_thread.start()
    def _playback_worker(self):
        """Background worker for playing queued audio"""
        while self.is_playing or not self.playback_queue.empty():
            try:
                item = self.playback_queue.get(timeout=0.5)
                item_type, data = item
                if item_type == 'sound':
                    self.play_sound(data, wait=True)
                self.playback_queue.task_done()
            except queue.Empty:
                continue
            except Exception as e:
                logger.error(f"Playback worker error: {e}")
        self.is_playing = False
    def cleanup(self):
        """Clean up audio resources"""
        self.is_playing = False
        self.stop_all_sounds()
        if self.playback_thread:
            self.playback_thread.join(timeout=2.0)
        pygame.mixer.quit()
        logger.info("Audio output cleanup complete")
class PiperTTSProvider:
    """Text-to-speech provider using Piper binary"""
    def __init__(self, config: dict):
        """
        Initialize Piper TTS provider
        Args:
            config: Configuration dictionary from settings.yaml
        """
        self.config = config
        self.piper_config = config['speech']['tts']['piper']
        self.piper_binary = self.piper_config['binary_path']
        self.model_path = self.piper_config['model_path']
        self.length_scale = self.piper_config.get('length_scale', 1.0)
        self.temp_dir = self.piper_config.get('temp_dir', '/tmp')
        if not Path(self.piper_binary).exists():
            raise FileNotFoundError(f"Piper binary not found: {self.piper_binary}")
        if not Path(self.model_path).exists():
            raise FileNotFoundError(f"Piper model not found: {self.model_path}")
        pygame.mixer.init()
        self.speech_queue = queue.Queue()
        self.is_speaking = False
        self.speech_thread: Optional[threading.Thread] = None
        self.current_channel = None
        logger.info(f"Piper TTS initialized with model: {self.model_path}")
    def speak(self, text: str, wait: bool = False):
        """
        Convert text to speech and play
        Args:
            text: Text to speak
            wait: If True, wait for speech to finish
        """
        try:
            if wait:
                self._synthesize_and_play(text, wait=True)
            else:
                self.speak_async(text)
            logger.info(f"Speaking: {text[:50]}...")
        except Exception as e:
            logger.error(f"Piper TTS error: {e}")
    def _synthesize_and_play(self, text: str, wait: bool = False):
        """
        Synthesize speech with Piper and play the audio
        Args:
            text: Text to synthesize
            wait: If True, wait for playback to finish
        """
        temp_wav = Path(self.temp_dir) / f"piper_{uuid.uuid4()}.wav"
        try:
            cmd = [
                self.piper_binary,
                '--model', self.model_path,
                '--length_scale', str(self.length_scale),
                '--output_file', str(temp_wav)
            ]
            subprocess.run(
                cmd,
                input=text,
                text=True,
                capture_output=True,
                check=True,
                timeout=10
            )
            if temp_wav.exists():
                sound = pygame.mixer.Sound(str(temp_wav))
                self.current_channel = sound.play()
                if wait and self.current_channel:
                    while self.current_channel.get_busy():
                        pygame.time.wait(100)
                if not wait:
                    threading.Timer(2.0, lambda: self._cleanup_wav(temp_wav)).start()
                else:
                    self._cleanup_wav(temp_wav)
        except subprocess.TimeoutExpired:
            logger.error("Piper synthesis timed out")
            self._cleanup_wav(temp_wav)
        except subprocess.CalledProcessError as e:
            logger.error(f"Piper synthesis failed: {e.stderr}")
            self._cleanup_wav(temp_wav)
        except Exception as e:
            logger.error(f"Error synthesizing speech: {e}")
            self._cleanup_wav(temp_wav)
    def _cleanup_wav(self, wav_path: Path):
        """Clean up temporary WAV file"""
        try:
            if wav_path.exists():
                wav_path.unlink()
        except Exception as e:
            logger.debug(f"Could not clean up WAV file: {e}")
    def speak_async(self, text: str):
        """
        Speak text asynchronously in background
        Args:
            text: Text to speak
        """
        self.speech_queue.put(text)
        if not self.is_speaking:
            self._start_speech_thread()
    def _start_speech_thread(self):
        """Start background thread for TTS"""
        if self.speech_thread and self.speech_thread.is_alive():
            return
        self.is_speaking = True
        self.speech_thread = threading.Thread(target=self._speech_worker)
        self.speech_thread.daemon = True
        self.speech_thread.start()
    def _speech_worker(self):
        """Background worker for TTS queue"""
        while self.is_speaking or not self.speech_queue.empty():
            try:
                text = self.speech_queue.get(timeout=0.5)
                self._synthesize_and_play(text, wait=True)
                self.speech_queue.task_done()
            except queue.Empty:
                continue
            except Exception as e:
                logger.error(f"Speech worker error: {e}")
        self.is_speaking = False
    def stop_speaking(self):
        """Stop current speech"""
        try:
            pygame.mixer.stop()
            while not self.speech_queue.empty():
                try:
                    self.speech_queue.get_nowait()
                except queue.Empty:
                    break
            logger.info("Speech stopped")
        except Exception as e:
            logger.error(f"Error stopping speech: {e}")
    def set_rate(self, rate: float):
        """
        Set speech rate (via length_scale)
        Args:
            rate: Speech rate multiplier (1.0 = normal, >1.0 = slower, <1.0 = faster)
        """
        self.length_scale = rate
    def set_volume(self, volume: float):
        """
        Set speech volume
        Args:
            volume: Volume level (0.0 to 1.0)
        """
        pygame.mixer.music.set_volume(volume)
    def cleanup(self):
        """Clean up TTS resources"""
        self.is_speaking = False
        self.stop_speaking()
        if self.speech_thread:
            self.speech_thread.join(timeout=2.0)
        pygame.mixer.quit()
        logger.info("Piper TTS cleanup complete")
class PyttxTTSProvider:
    """Text-to-speech provider using pyttsx3"""
    def __init__(self, config: dict):
        """
        Initialize TTS engine
        Args:
            config: Configuration dictionary from settings.yaml
        """
        self.config = config
        self.tts_config = config['speech']['tts']['pyttsx3']
        self.engine = pyttsx3.init()
        self.engine.setProperty('rate', self.tts_config['rate'])
        self.engine.setProperty('volume', self.tts_config['volume'])
        voices = self.engine.getProperty('voices')
        voice_id = self.tts_config.get('voice_id', 0)
        if 0 <= voice_id < len(voices):
            self.engine.setProperty('voice', voices[voice_id].id)
        self.speech_queue = queue.Queue()
        self.is_speaking = False
        self.speech_thread: Optional[threading.Thread] = None
        logger.info("TTS initialized")
    def speak(self, text: str, wait: bool = False):
        """
        Convert text to speech and play
        Args:
            text: Text to speak
            wait: If True, wait for speech to finish
        """
        try:
            if wait:
                self.engine.say(text)
                self.engine.runAndWait()
            else:
                self.speak_async(text)
            logger.info(f"Speaking: {text}")
        except Exception as e:
            logger.error(f"TTS error: {e}")
    def speak_async(self, text: str):
        """
        Speak text asynchronously in background
        Args:
            text: Text to speak
        """
        self.speech_queue.put(text)
        if not self.is_speaking:
            self._start_speech_thread()
    def _start_speech_thread(self):
        """Start background thread for TTS"""
        if self.speech_thread and self.speech_thread.is_alive():
            return
        self.is_speaking = True
        self.speech_thread = threading.Thread(target=self._speech_worker)
        self.speech_thread.daemon = True
        self.speech_thread.start()
    def _speech_worker(self):
        """Background worker for TTS queue"""
        while self.is_speaking or not self.speech_queue.empty():
            try:
                text = self.speech_queue.get(timeout=0.5)
                self.engine.say(text)
                self.engine.runAndWait()
                self.speech_queue.task_done()
            except queue.Empty:
                continue
            except Exception as e:
                logger.error(f"Speech worker error: {e}")
        self.is_speaking = False
    def stop_speaking(self):
        """Stop current speech"""
        try:
            self.engine.stop()
            while not self.speech_queue.empty():
                try:
                    self.speech_queue.get_nowait()
                except queue.Empty:
                    break
            logger.info("Speech stopped")
        except Exception as e:
            logger.error(f"Error stopping speech: {e}")
    def set_rate(self, rate: int):
        """
        Set speech rate
        Args:
            rate: Words per minute
        """
        self.engine.setProperty('rate', rate)
    def set_volume(self, volume: float):
        """
        Set speech volume
        Args:
            volume: Volume level (0.0 to 1.0)
        """
        self.engine.setProperty('volume', volume)
    def list_voices(self):
        """List available TTS voices"""
        voices = self.engine.getProperty('voices')
        voice_list = []
        for i, voice in enumerate(voices):
            voice_list.append({
                'index': i,
                'id': voice.id,
                'name': voice.name,
                'languages': voice.languages
            })
        return voice_list
    def cleanup(self):
        """Clean up TTS resources"""
        self.is_speaking = False
        self.stop_speaking()
        if self.speech_thread:
            self.speech_thread.join(timeout=2.0)
        logger.info("TTS cleanup complete")
class TextToSpeech:
    """
    TTS Factory - Creates appropriate TTS provider based on configuration
    Maintains backward compatibility with existing code
    """
    def __init__(self, config: dict):
        """
        Initialize TTS with configured provider
        Args:
            config: Configuration dictionary from settings.yaml
        """
        self.config = config
        provider_name = config['speech']['tts'].get('provider', 'pyttsx3')
        if provider_name == 'piper':
            logger.info("Initializing Piper TTS provider")
            self.provider = PiperTTSProvider(config)
            self.engine = None
        elif provider_name == 'pyttsx3':
            logger.info("Initializing pyttsx3 TTS provider")
            self.provider = PyttxTTSProvider(config)
            self.engine = self.provider.engine
        else:
            raise ValueError(f"Unknown TTS provider: {provider_name}. Use 'piper' or 'pyttsx3'")
        self.provider_name = provider_name
    def speak(self, text: str, wait: bool = False):
        """Delegate to provider"""
        return self.provider.speak(text, wait)
    def speak_async(self, text: str):
        """Delegate to provider"""
        return self.provider.speak_async(text)
    def stop_speaking(self):
        """Delegate to provider"""
        return self.provider.stop_speaking()
    def set_rate(self, rate: float):
        """Delegate to provider (behavior differs by provider)"""
        if hasattr(self.provider, 'set_rate'):
            return self.provider.set_rate(rate)
    def set_volume(self, volume: float):
        """Delegate to provider"""
        if hasattr(self.provider, 'set_volume'):
            return self.provider.set_volume(volume)
    def list_voices(self):
        """List available voices (pyttsx3 only)"""
        if hasattr(self.provider, 'list_voices'):
            return self.provider.list_voices()
        else:
            logger.warning(f"{self.provider_name} does not support list_voices()")
            return []
    def cleanup(self):
        """Delegate to provider"""
        return self.provider.cleanup()
    @property
    def is_speaking(self):
        """Check if currently speaking"""
        return self.provider.is_speaking
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    config = {
        'audio': {
            'output': {
                'sample_rate': 22050,
                'channels': 1
            }
        },
        'speech': {
            'tts': {
                'pyttsx3': {
                    'rate': 150,
                    'volume': 0.9,
                    'voice_id': 0,
                    'pitch': 1.5
                }
            }
        }
    }
    print("Testing Audio Output...")
    audio_output = AudioOutput(config)
    audio_output.set_volume(0.7)
    print("\nTesting TTS...")
```

```
tts = TextToSpeech(config)
print("Available voices:")
for voice in tts.list_voices():
    print(f"  [{voice['index']}] {voice['name']}")
print("\nSpeaking test...")
tts.speak("Hello! I'm your companion bot. I love to play and learn!", wait=True)
print("\nAsync speech test...")
tts.speak_async("This is asynchronous speech.")
tts.speak_async("I can queue multiple sentences.")
import time
time.sleep(5)
print("\nCleaning up...")
audio_output.cleanup()
tts.cleanup()
print("Test complete!")
```

```
### src/audio/voice_detector.py
"""
Voice Activity Detector
Enhanced voice detection for mini microphone
"""
import numpy as np
import webrtcvad
import logging
from collections import deque
from scipy import signal
logger = logging.getLogger(__name__)
class VoiceActivityDetector:
    """Advanced voice activity detection with noise filtering"""
    def __init__(self, config: dict):
        """
        Initialize voice activity detector
        Args:
            config: Audio configuration dictionary
        """
        self.config = config
        self.vad_config = config['audio']['processing']
        self.sample_rate = config['audio']['input']['sample_rate']
        self.vad = webrtcvad.Vad(self.vad_config['vad_aggressiveness'])
        self.noise_floor = self.vad_config['silence_threshold']
        self.noise_floor_samples = deque(maxlen=100)
        self.is_voice_active = False
        self.voice_frames = 0
        self.silence_frames = 0
        self.min_voice_frames = 3
        self.max_silence_frames = 10
    def detect(self, audio_chunk: bytes) -> bool:
        """
        Detect if audio chunk contains voice activity
        Args:
            audio_chunk: Raw audio bytes (int16)
        Returns:
            True if voice is detected, False otherwise
        """
        audio_array = np.frombuffer(audio_chunk, dtype=np.int16)
        amplitude = np.abs(audio_array).mean()
        self._update_noise_floor(amplitude)
        amplitude_threshold = max(self.noise_floor * 1.2, 150)
        amplitude_check = amplitude > amplitude_threshold
        vad_check = self._check_webrtc_vad(audio_chunk)
        is_voice = amplitude_check and vad_check
        if not hasattr(self, '_debug_counter'):
            self._debug_counter = 0
        self._debug_counter += 1
        if self._debug_counter % 50 == 0:
            logger.debug(f"VAD: amp={amplitude:.0f}, threshold={amplitude_threshold:.0f}, "
                        f"amp_ok={amplitude_check}, vad_ok={vad_check}, voice={is_voice}")
        if is_voice:
            self.voice_frames += 1
            self.silence_frames = 0
            if self.voice_frames >= self.min_voice_frames:
                self.is_voice_active = True
        else:
            self.silence_frames += 1
            self.voice_frames = 0
            if self.silence_frames >= self.max_silence_frames:
                self.is_voice_active = False
        return self.is_voice_active
    def _update_noise_floor(self, amplitude: float):
        """
        Update adaptive noise floor estimate
        Args:
            amplitude: Current amplitude value
        """
        self.noise_floor_samples.append(amplitude)
        if len(self.noise_floor_samples) >= 10:
            self.noise_floor = np.median(list(self.noise_floor_samples))
    def _check_webrtc_vad(self, audio_chunk: bytes) -> bool:
        """
        Check voice activity using WebRTC VAD
        Args:
            audio_chunk: Raw audio bytes
        Returns:
            True if voice detected by WebRTC VAD
        """
        try:
            audio_array = np.frombuffer(audio_chunk, dtype=np.int16)
            target_rate = 16000
            if self.sample_rate != target_rate:
                num_samples = int(len(audio_array) * target_rate / self.sample_rate)
                audio_array = signal.resample(audio_array, num_samples).astype(np.int16)
                vad_sample_rate = target_rate
            else:
                vad_sample_rate = self.sample_rate
            frame_duration = 30
            frame_size = int(vad_sample_rate * frame_duration / 1000)
            if len(audio_array) < frame_size:
                audio_array = np.pad(audio_array, (0, frame_size - len(audio_array)))
            elif len(audio_array) > frame_size:
                audio_array = audio_array[:frame_size]
            audio_bytes = audio_array.tobytes()
            return self.vad.is_speech(audio_bytes, vad_sample_rate)
        except Exception as e:
            logger.debug(f"WebRTC VAD error: {e}")
            return False
    def reset(self):
        """Reset voice detection state"""
        self.is_voice_active = False
        self.voice_frames = 0
        self.silence_frames = 0
        self.noise_floor_samples.clear()
        logger.debug("Voice detector reset")
    def get_confidence(self) -> float:
        """
        Get voice detection confidence (0-1)
        Returns:
            Confidence score
        """
        if not self.is_voice_active:
            return 0.0
        confidence = min(1.0, self.voice_frames / (self.min_voice_frames * 3))
        return confidence
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    config = {
        'audio': {
            'input': {
                'sample_rate': 16000,
                'chunk_size': 1024
            },
            'processing': {
                'vad_aggressiveness': 2,
                'silence_threshold': 500
            }
        }
    }
    vad = VoiceActivityDetector(config)
    print("Voice Activity Detector initialized")
    print("Test with actual audio input to see results")
```

### src/expression/__init__.py

```python
"""
Expression Module
Manages visual expressions and physical movements
"""
from .emotion_display import EmotionDisplay
from .display_renderer import DisplayRenderer
from .transition_controller import TransitionController
__all__ = [
    'EmotionDisplay',
    'DisplayRenderer',
    'TransitionController'
]
```

### src/expression/display_renderer.py

```
"""
Display Renderer for Emotion Expression Pipeline
Handles pygame initialization, image loading, and frame rendering for piTFT
"""
try:
    import pygame  # type: ignore  # noqa: E0401
except ImportError:
    pygame = None
import os
import glob
import logging
from pathlib import Path
from typing import Dict, Tuple, Optional, Any
from .procedural_face import ProceduralFaceRenderer
logger = logging.getLogger(__name__)
EmotionFrames = Tuple[pygame.Surface, pygame.Surface]
class DisplayRenderer:
    """
    Pygame rendering engine for emotion display
    Manages piTFT framebuffer, image loading, and alpha blending
    """
    def __init__(
        self,
        screen_size: Tuple[int, int] = (320, 240),
        framebuffer: str = "/dev/fb0",
        image_dir: str = "src/display",
        procedural_config: Optional[Dict[str, Any]] = None,
    ):
        """
        Initialize display renderer
        Args:
            screen_size: Display resolution (width, height)
            framebuffer: Path to framebuffer device (piTFT)
            image_dir: Directory containing emotion sprite images
        """
        self.screen_size = screen_size
        self.framebuffer = framebuffer
        self.image_dir = Path(image_dir)
        self.screen: Optional[pygame.Surface] = None
        self.emotion_images: Dict[str, EmotionFrames] = {}
        self.listening_image: Optional[pygame.Surface] = None
        procedural_config = procedural_config or {}
        self.use_procedural = procedural_config.get('enabled', False)
        self.procedural_renderer: Optional[ProceduralFaceRenderer] = None
        self._init_pygame()
        if self.use_procedural:
            self.procedural_renderer = ProceduralFaceRenderer(
                self.screen_size, procedural_config
            )
            logger.info("Procedural face renderer enabled")
        else:
            self._load_emotion_images()
    def _init_pygame(self):
        """Initialize pygame with piTFT framebuffer or fallback to window"""
        try:
            os.putenv('SDL_VIDEODRIVER', 'fbcon')
            os.putenv('SDL_FBDEV', self.framebuffer)
            os.putenv('SDL_NOMOUSE', '1')
            pygame.init()
            pygame.mouse.set_visible(False)
            self.screen = pygame.display.set_mode(self.screen_size)
            logger.info("Initialized piTFT display at %s", self.framebuffer)
        except pygame.error as e:
            logger.warning("piTFT initialization failed: %s", e)
            logger.warning("Falling back to window mode for testing")
            if 'SDL_VIDEODRIVER' in os.environ:
                del os.environ['SDL_VIDEODRIVER']
            if 'SDL_FBDEV' in os.environ:
                del os.environ['SDL_FBDEV']
            pygame.init()
            self.screen = pygame.display.set_mode(self.screen_size)
            pygame.display.set_caption("Emotion Display")
            logger.info("Initialized window mode")
    def _load_emotion_images(self):
        """
        Load all emotion sprite images from directory
        Automatically pairs base and speaking frames
        Pattern from test_emotion_display.py lines 10-41
        """
        if not self.image_dir.exists():
            logger.error("Image directory not found: %s", self.image_dir)
            return
        base_files = glob.glob(str(self.image_dir / "*.png"))
        loaded_count = 0
        for base_path in base_files:
            filename = os.path.basename(base_path)
            if "_speaking" in filename or "_active" in filename:
                continue
            emotion_name = os.path.splitext(filename)[0]
            if emotion_name == "listening":
                self._load_listening_image(base_path)
                continue
            try:
                base_surface = self._load_and_scale_image(base_path)
                speaking_path = self.image_dir / f"{emotion_name}_speaking.png"
                if speaking_path.exists():
                    speaking_surface = self._load_and_scale_image(
                        str(speaking_path)
                    )
                else:
                    speaking_surface = base_surface
                    logger.warning(
                        "No speaking frame for %s; using base", emotion_name
                    )
                self.emotion_images[emotion_name] = (
                    base_surface,
                    speaking_surface,
                )
                loaded_count += 1
                logger.debug("Loaded emotion: %s", emotion_name)
            except pygame.error as e:
                logger.error("Failed to load %s: %s", emotion_name, e)
        logger.info("Loaded %s emotion pairs", loaded_count)
        if loaded_count == 0:
            logger.error("No emotion images loaded; display disabled.")
    def _load_listening_image(self, base_path: str):
        """Load the special listening state image"""
        try:
            self.listening_image = self._load_and_scale_image(base_path)
            logger.debug("Loaded listening state image")
        except pygame.error as e:
            logger.error("Failed to load listening image: %s", e)
    def _load_and_scale_image(self, image_path: str) -> pygame.Surface:
        """
        Load PNG image and scale to screen size if needed
        Args:
            image_path: Path to PNG file
        Returns:
            Scaled pygame surface with alpha channel
        """
        surface = pygame.image.load(image_path).convert_alpha()
        if surface.get_size() != self.screen_size:
            surface = pygame.transform.smoothscale(surface, self.screen_size)
        return surface
    def get_emotion_frame(
        self, emotion: str, speaking: bool = False
    ) -> Optional[pygame.Surface]:
        """
        Get emotion surface for rendering
        Args:
            emotion: Emotion name (e.g., 'happy', 'sad')
            speaking: If True, return speaking frame; else base frame
        Returns:
            Pygame surface or None if emotion not found
        """
        if emotion not in self.emotion_images:
            logger.warning(
                "Emotion '%s' not found, using 'happy' fallback", emotion
            )
            emotion = 'happy'
            if emotion not in self.emotion_images:
                logger.error("Fallback emotion 'happy' also missing!")
                return None
        base_frame, speaking_frame = self.emotion_images[emotion]
        return speaking_frame if speaking else base_frame
    def get_listening_frame(self) -> Optional[pygame.Surface]:
        """Get listening state surface"""
        if self.listening_image is None:
            logger.warning("Listening image not loaded")
        return self.listening_image
    def render_frame(self, surface: pygame.Surface):
        """
        Render surface to screen
        Args:
            surface: Pygame surface to display
        """
        if surface is None:
            logger.warning("Attempted to render None surface")
            return
        self.screen.blit(surface, (0, 0))
        pygame.display.flip()
    def render_procedural(
        self,
        current_params: Dict,
        target_params: Optional[Dict],
        blend_alpha: float,
        speaking_level: float,
        listening: bool,
        delta_time: float,
    ):
        if not self.use_procedural or not self.procedural_renderer:
            logger.warning("Procedural renderer not available")
            return
        self.procedural_renderer.update_state(
            delta_time, speaking_level, listening
        )
        frame = self.procedural_renderer.render(
            current_params=current_params,
            target_params=target_params,
            blend_alpha=blend_alpha,
            listening=listening,
        )
        self.render_frame(frame)
    def create_blended_frame(
        self,
        img1: pygame.Surface,
        img2: pygame.Surface,
        alpha: float,
    ) -> pygame.Surface:
        """
        Create alpha-blended composite of two images for smooth transitions
        Args:
            img1: Current emotion surface
            img2: Target emotion surface
            alpha: Blend factor (0.0 = all img1, 1.0 = all img2)
        Returns:
            Blended surface
        """
        if img1 is None or img2 is None:
            logger.error("Cannot blend None surfaces")
            return img1 if img1 is not None else img2
        alpha = max(0.0, min(1.0, alpha))
        result = pygame.Surface(self.screen_size, pygame.SRCALPHA)
        img1_copy = img1.copy()
        img1_copy.set_alpha(int(255 * (1 - alpha)))
        result.blit(img1_copy, (0, 0))
        img2_copy = img2.copy()
        img2_copy.set_alpha(int(255 * alpha))
        result.blit(img2_copy, (0, 0))
        return result
    def clear_screen(self, color: Tuple[int, int, int] = (0, 0, 0)):
        """
        Clear screen with solid color
        Args:
            color: RGB color tuple (default: black)
        """
        self.screen.fill(color)
    def cleanup(self):
        """Clean up pygame resources"""
        pygame.quit()
        logger.info("Display renderer cleanup complete")
```

### src/expression/emotion_display.py

```
"""
Emotion Display Controller for Companion Bot
Main controller for expression pipeline with threading and state management
"""
try:
    import pygame  # type: ignore  # noqa: E0401
except ImportError:
    pygame = None
import threading
import queue
import time
import logging
import math
from typing import Optional, Dict, Callable, Tuple
try:
    import RPi.GPIO as GPIO
    GPIO_AVAILABLE = True
except ImportError:
    GPIO_AVAILABLE = False
    logging.warning("RPi.GPIO not available - GPIO features disabled")
from .display_renderer import DisplayRenderer
from .transition_controller import TransitionController
logger = logging.getLogger(__name__)
DEFAULT_EMOTION_PRESETS: Dict[str, Dict] = {
    "happy": {
        "eye_spacing": 90,
        "eye_width": 44,
        "eye_height": 38,
        "eye_y": 95,
        "pupil_size": 12,
        "pupil_offset": 2,
        "brow_raise": 0.1,
        "brow_slant": -0.05,
        "mouth_width": 130,
        "mouth_height": 26,
        "mouth_curve": 0.35,
        "mouth_open": 0.08,
        "mouth_sensitivity": 0.7,
        "eye_color": [245, 245, 245],
        "pupil_color": [20, 20, 20],
        "brow_color": [230, 230, 230],
        "mouth_color": [240, 140, 140],
    },
    "sad": {
        "eye_spacing": 85,
        "eye_width": 42,
        "eye_height": 36,
        "eye_y": 100,
        "pupil_size": 11,
        "pupil_offset": -1,
        "brow_raise": -0.1,
        "brow_slant": 0.15,
        "mouth_width": 120,
        "mouth_height": 22,
        "mouth_curve": -0.45,
        "mouth_open": 0.05,
        "mouth_sensitivity": 0.55,
        "eye_color": [235, 235, 240],
        "pupil_color": [15, 15, 25],
        "brow_color": [210, 210, 220],
        "mouth_color": [200, 120, 160],
    },
    "excited": {
        "eye_spacing": 95,
        "eye_width": 46,
        "eye_height": 40,
        "eye_y": 92,
        "pupil_size": 13,
        "pupil_offset": 3,
        "brow_raise": 0.25,
        "brow_slant": -0.08,
        "mouth_width": 140,
        "mouth_height": 28,
        "mouth_curve": 0.5,
        "mouth_open": 0.18,
        "mouth_sensitivity": 0.9,
        "eye_color": [250, 250, 250],
        "pupil_color": [10, 10, 10],
        "brow_color": [240, 240, 240],
        "mouth_color": [255, 150, 150],
    },
    "curious": {
        "eye_spacing": 88,
        "eye_width": 44,
        "eye_height": 38,
        "eye_y": 96,
        "pupil_size": 12,
        "pupil_offset": 4,
        "brow_raise": 0.05,
        "brow_slant": -0.15,
        "mouth_width": 115,
        "mouth_height": 22,
        "mouth_curve": 0.12,
        "mouth_open": 0.06,
        "mouth_sensitivity": 0.6,
        "eye_color": [245, 245, 245],
        "pupil_color": [25, 25, 25],
        "brow_color": [225, 225, 225],
        "mouth_color": [220, 150, 160],
    },
    "sleepy": {
        "eye_spacing": 85,
        "eye_width": 44,
        "eye_height": 24,
        "eye_y": 104,
        "pupil_size": 10,
        "pupil_offset": 0,
        "brow_raise": -0.05,
        "brow_slant": 0.05,
        "mouth_width": 110,
        "mouth_height": 18,
        "mouth_curve": -0.1,
        "mouth_open": 0.04,
        "mouth_sensitivity": 0.4,
        "eye_color": [235, 235, 235],
        "pupil_color": [20, 20, 20],
        "brow_color": [215, 215, 215],
        "mouth_color": [200, 140, 150],
    },
    "angry": {
        "eye_spacing": 90,
        "eye_width": 44,
        "eye_height": 34,
        "eye_y": 94,
        "pupil_size": 12,
        "pupil_offset": -2,
        "brow_raise": -0.15,
        "brow_slant": 0.25,
        "mouth_width": 125,
        "mouth_height": 24,
        "mouth_curve": -0.25,
        "mouth_open": 0.09,
        "mouth_sensitivity": 0.75,
        "eye_color": [240, 230, 230],
        "pupil_color": [30, 10, 10],
        "brow_color": [220, 200, 200],
        "mouth_color": [255, 120, 120],
    },
    "scared": {
        "eye_spacing": 90,
        "eye_width": 46,
        "eye_height": 42,
        "eye_y": 92,
        "pupil_size": 10,
        "pupil_offset": -3,
        "brow_raise": 0.2,
        "brow_slant": 0.1,
        "mouth_width": 125,
        "mouth_height": 22,
        "mouth_curve": -0.35,
        "mouth_open": 0.14,
        "mouth_sensitivity": 0.85,
        "eye_color": [240, 240, 245],
        "pupil_color": [15, 15, 20],
        "brow_color": [225, 225, 235],
        "mouth_color": [240, 150, 180],
    },
    "playful": {
        "eye_spacing": 94,
        "eye_width": 44,
        "eye_height": 38,
        "eye_y": 94,
        "pupil_size": 13,
        "pupil_offset": 5,
        "brow_raise": 0.1,
        "brow_slant": -0.12,
        "mouth_width": 135,
        "mouth_height": 26,
        "mouth_curve": 0.28,
        "mouth_open": 0.12,
        "mouth_sensitivity": 0.8,
        "eye_color": [250, 250, 250],
        "pupil_color": [15, 15, 15],
        "brow_color": [235, 235, 235],
        "mouth_color": [255, 170, 140],
    },
    "lonely": {
        "eye_spacing": 82,
        "eye_width": 42,
        "eye_height": 34,
        "eye_y": 100,
        "pupil_size": 11,
        "pupil_offset": -1,
        "brow_raise": -0.08,
        "brow_slant": 0.1,
        "mouth_width": 112,
        "mouth_height": 20,
        "mouth_curve": -0.2,
        "mouth_open": 0.05,
        "mouth_sensitivity": 0.6,
        "eye_color": [235, 235, 240],
        "pupil_color": [15, 15, 20],
        "brow_color": [215, 215, 225],
        "mouth_color": [205, 140, 170],
    },
    "bored": {
        "eye_spacing": 90,
        "eye_width": 42,
        "eye_height": 30,
        "eye_y": 102,
        "pupil_size": 11,
        "pupil_offset": 0,
        "brow_raise": -0.02,
        "brow_slant": 0.0,
        "mouth_width": 118,
        "mouth_height": 18,
        "mouth_curve": -0.05,
        "mouth_open": 0.04,
        "mouth_sensitivity": 0.45,
        "eye_color": [240, 240, 240],
        "pupil_color": [20, 20, 20],
        "brow_color": [220, 220, 220],
        "mouth_color": [200, 140, 150],
    },
    "surprised": {
        "eye_spacing": 92,
        "eye_width": 46,
        "eye_height": 44,
        "eye_y": 92,
        "pupil_size": 11,
        "pupil_offset": 0,
        "brow_raise": 0.18,
        "brow_slant": 0.02,
        "mouth_width": 118,
        "mouth_height": 26,
        "mouth_curve": 0.12,
        "mouth_open": 0.2,
        "mouth_sensitivity": 0.9,
        "eye_color": [250, 250, 250],
        "pupil_color": [15, 15, 15],
        "brow_color": [240, 240, 240],
        "mouth_color": [240, 160, 180],
    },
    "loving": {
        "eye_spacing": 90,
        "eye_width": 44,
        "eye_height": 38,
        "eye_y": 95,
        "pupil_size": 12,
        "pupil_offset": 3,
        "brow_raise": 0.15,
        "brow_slant": -0.05,
        "mouth_width": 130,
        "mouth_height": 26,
        "mouth_curve": 0.32,
        "mouth_open": 0.12,
        "mouth_sensitivity": 0.75,
        "eye_color": [245, 245, 245],
        "pupil_color": [20, 20, 20],
        "brow_color": [230, 230, 230],
        "mouth_color": [255, 150, 170],
    },
}
class DisplayState:
    """Container for display state"""
    def __init__(self):
        self.is_listening: bool = False
        self.is_speaking: bool = False
        self.current_emotion: str = "happy"
        self.target_emotion: Optional[str] = None
        self.speaking_frame_toggle: bool = False
        self.last_toggle_time: float = 0.0
        self.toggle_interval: float = 0.15
        self.speaking_level: float = 0.0
        self.speaking_level_target: float = 0.0
        self.speaking_phase: float = 0.0
        self.last_gesture_time: float = 0.0
        self.pending_tap_time: float = 0.0
        self.gesture_busy_until: float = 0.0
        self.petting_active: bool = False
class EmotionDisplay:
    """
    Main emotion display controller
    Manages display lifecycle, state machine, threading, and GPIO
    """
    def __init__(self, config: dict, framebuffer: str = "/dev/fb0"):
        """
        Initialize emotion display
        Args:
            config: Configuration dictionary from settings.yaml
            framebuffer: Framebuffer device path (default: /dev/fb0 for piTFT)
        """
        self.config = config
        self.display_config = config.get('expression', {}).get('display', {})
        self.procedural_config = self.display_config.get('procedural_face', {})
        self.speaking_rest_factor = self.procedural_config.get(
            'speaking_rest_factor', 0.35
        )
        self.speaking_wave_hz = self.procedural_config.get(
            'speaking_wave_hz', 6.0
        )
        self.touch_config = self.display_config.get('touch', {})
        self.touch_enabled = self.touch_config.get('enabled', True)
        self.touch_thresholds = self.touch_config.get('thresholds', {})
        self.gesture_effects = self.touch_config.get('gesture_effects', {})
        self.gesture_cooldown = float(
            self.touch_thresholds.get('cooldown', 0.8)
        )
        self.effect_queue_cooldown = float(
            self.touch_thresholds.get('effect_cooldown', 0.4)
        )
        self.effect_busy_window = float(
            self.touch_thresholds.get('effect_busy', 1.2)
        )
        self.effect_callback: Optional[Callable[[Dict], None]] = None
        self.exit_callback: Optional[Callable[[], None]] = None
        screen_size = tuple(self.display_config.get('resolution', [320, 240]))
        image_dir = self.display_config.get('image_dir', 'src/display')
        self.fps = self.display_config.get('fps', 60)
        gpio_cfg = self.display_config.get('gpio', {})
        self.gpio_enabled = gpio_cfg.get('enabled', True)
        self.gpio_exit_pin = gpio_cfg.get('exit_button_pin', 27)
        self.renderer = DisplayRenderer(
            screen_size=screen_size,
            framebuffer=framebuffer,
            image_dir=image_dir,
            procedural_config=self.procedural_config
        )
        self.transition = TransitionController()
        self.state = DisplayState()
        speaking_cfg = self.display_config.get('speaking', {})
        self.state.toggle_interval = speaking_cfg.get('toggle_interval', 0.15)
        self.procedural_enabled = self.renderer.use_procedural
        self.emotion_params: Dict[str, Dict] = self._build_emotion_params()
        self.is_running = False
        self.display_thread: Optional[threading.Thread] = None
        self.command_queue = queue.Queue()
        self.state_lock = threading.Lock()
        self.clock = pygame.time.Clock()
        self._last_delta_time = 0.0
        self._touch_start_pos: Optional[Tuple[int, int]] = None
        self._touch_down_pos: Optional[Tuple[int, int]] = None
        self._touch_down_time: float = 0.0
        self._last_tap_time: float = 0.0
        self._drag_distance: float = 0.0
        self._last_effect_time: float = 0.0
        self.pending_tap_time: float = 0.0
        self._init_gpio()
        logger.info("EmotionDisplay initialized")
    def _init_gpio(self):
        """Initialize GPIO for exit button (if available)"""
        if not GPIO_AVAILABLE or not self.gpio_enabled:
            logger.info("GPIO disabled")
            return
        try:
            GPIO.setmode(GPIO.BCM)
            GPIO.setup(self.gpio_exit_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
            logger.info(
                "GPIO initialized - exit button on pin %s",
                self.gpio_exit_pin
            )
        except (RuntimeError, ValueError, OSError) as e:
            logger.error("GPIO initialization failed: %s", e)
    def start(self):
        """Start the display loop in a dedicated thread"""
        if self.is_running:
            logger.warning("Display already running")
            return
        self.is_running = True
        self.display_thread = threading.Thread(
            target=self._display_loop,
            daemon=True
        )
        self.display_thread.start()
        logger.info("Display thread started")
    def stop(self):
        """Stop the display loop gracefully"""
        if not self.is_running:
            return
        self.is_running = False
        if self.display_thread:
            self.display_thread.join(timeout=2.0)
        logger.info("Display thread stopped")
    def _display_loop(self):
        """
        Main display loop - runs at configured FPS (default 60)
        Pattern from camera.py and two_collide.py
        """
        logger.info("Display loop starting at %s FPS", self.fps)
        last_time = time.time()
        while self.is_running:
            current_time = time.time()
            delta_time = current_time - last_time
            last_time = current_time
            self._last_delta_time = delta_time
            self._process_commands()
            if self._check_gpio_exit():
                logger.info("GPIO exit button pressed")
                if self.exit_callback:
                    try:
                        self.exit_callback()
                    except Exception as exc:
                        logger.error("Exit callback failed: %s", exc)
                self.is_running = False
                break
            self._update_state(delta_time)
            self._render_frame()
            if self.touch_enabled and self.state.petting_active:
                self._discard_touch_events()
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.is_running = False
                elif self.touch_enabled:
                    self._handle_touch_event(event)
            self.clock.tick(self.fps)
        logger.info("Display loop exited")
    def _process_commands(self):
        """Process commands from the command queue"""
        try:
            while True:
                cmd = self.command_queue.get_nowait()
                self._execute_command(cmd)
        except queue.Empty:
            pass
    def _execute_command(self, cmd: dict):
        """
        Execute a command
        Args:
            cmd: Command dictionary with 'type' and parameters
        """
        cmd_type = cmd.get('type')
        with self.state_lock:
            if cmd_type == 'SET_EMOTION':
                emotion = cmd.get('emotion', 'happy')
                duration = cmd.get('duration', 0.5)
                self._start_emotion_transition(emotion, duration)
            elif cmd_type == 'SET_LISTENING':
                active = cmd.get('active', False)
                self.state.is_listening = active
                logger.debug("Listening: %s", active)
            elif cmd_type == 'SET_SPEAKING':
                active = cmd.get('active', False)
                self.state.is_speaking = active
                level = cmd.get('level')
                if level is None:
                    level = 1.0 if active else 0.0
                self.state.speaking_level_target = max(
                    0.0, min(1.0, float(level))
                )
                if not active:
                    self.state.speaking_frame_toggle = False
                    self.state.speaking_phase = 0.0
                else:
                    self.state.speaking_phase = 0.0
                logger.debug("Speaking: %s", active)
            elif cmd_type == 'APPLY_EFFECT':
```

```
            effect = cmd.get('effect', {})
            self._apply_effect(effect)
        elif cmd_type == 'SET_PETTING':
            active = cmd.get('active', False)
            self.state.petting_active = bool(active)
def _start_emotion_transition(self, emotion: str, duration: float):
    """Start transition to new emotion"""
    if (
        emotion == self.state.current_emotion
        and not self.transition.is_transitioning()
    ):
        logger.debug("Already showing %s, skipping transition", emotion)
        return
    self.transition.start_transition(
        from_emotion=self.state.current_emotion,
        to_emotion=emotion,
        duration=duration
    )
    self.state.target_emotion = emotion
def _update_state(self, delta_time: float):
    """
    Update display state each frame
    Args:
        delta_time: Time since last update (seconds)
    """
    pending_tap_fire = False
    with self.state_lock:
        if self.transition.is_transitioning():
            _from_em, to_em, _alpha = self.transition.update(delta_time)
            if not self.transition.is_transitioning():
                self.state.current_emotion = to_em
                self.state.target_emotion = None
        if self.state.is_speaking or self.state.is_listening:
            current_time = time.time()
            if (
                current_time - self.state.last_toggle_time
                >= self.state.toggle_interval
            ):
                self.state.speaking_frame_toggle = (
                    not self.state.speaking_frame_toggle
                )
                self.state.last_toggle_time = current_time
        if self.procedural_enabled:
            smooth = max(
                1e-3, self.procedural_config.get('speaking_smooth', 8.0)
            )
            blend = 1.0 - math.exp(-smooth * max(delta_time, 0.0))
            target_level = 0.0
            if self.state.is_speaking:
                self.state.speaking_phase += (
                    2.0
                    * math.pi
                    * self.speaking_wave_hz
                    * max(delta_time, 0.0)
                )
                wave = 0.5 + 0.5 * math.sin(self.state.speaking_phase)
                target_level = self.state.speaking_level_target * (
                    self.speaking_rest_factor
                    + (1 - self.speaking_rest_factor) * wave
                )
            self.state.speaking_level = (
                (1 - blend) * self.state.speaking_level
                + blend * target_level
            )
    if self.touch_enabled and self.pending_tap_time > 0.0:
        double_tap_window = float(
            self.touch_thresholds.get('double_tap_window', 0.35)
        )
        now = time.time()
        if now - self.pending_tap_time > double_tap_window:
            pending_tap_fire = True
            self.pending_tap_time = 0.0
    if pending_tap_fire:
        self._trigger_gesture_effect("tap")
def _render_frame(self):
    """
    Render current frame based on state priority
    Priority: LISTENING > SPEAKING > EMOTION + TRANSITION
    """
    with self.state_lock:
        if self.state.is_listening:
            if self.procedural_enabled:
                params = self._get_emotion_params(
                    self.state.current_emotion
                )
                self.renderer.render_procedural(
                    current_params=params,
                    target_params=None,
                    blend_alpha=0.0,
                    speaking_level=0.0,
                    listening=True,
                    delta_time=self._last_delta_time
                )
                return
            frame = self.renderer.get_listening_frame()
            if frame:
                self.renderer.render_frame(frame)
                return
        if self.state.is_speaking:
            emotion = self.state.current_emotion
            if self.procedural_enabled:
                params = self._get_emotion_params(emotion)
                self.renderer.render_procedural(
                    current_params=params,
                    target_params=None,
                    blend_alpha=0.0,
                    speaking_level=max(0.1, self.state.speaking_level),
                    listening=False,
                    delta_time=self._last_delta_time
                )
                return
            frame = self.renderer.get_emotion_frame(
                emotion, speaking=self.state.speaking_frame_toggle
            )
            if frame:
                self.renderer.render_frame(frame)
                return
        if self.transition.is_transitioning():
            from_em, to_em, alpha = self.transition.update(0)
            if self.procedural_enabled:
                from_params = self._get_emotion_params(from_em)
                to_params = self._get_emotion_params(to_em)
                self.renderer.render_procedural(
                    current_params=from_params,
                    target_params=to_params,
                    blend_alpha=alpha,
                    speaking_level=0.0,
                    listening=False,
                    delta_time=self._last_delta_time
                )
                return
            from_frame = self.renderer.get_emotion_frame(
                from_em, speaking=False
            )
            to_frame = self.renderer.get_emotion_frame(
                to_em, speaking=False
            )
            if from_frame and to_frame:
                blended = self.renderer.create_blended_frame(
                    from_frame, to_frame, alpha
                )
                self.renderer.render_frame(blended)
                return
        if self.procedural_enabled:
            params = self._get_emotion_params(
                self.state.current_emotion
            )
            self.renderer.render_procedural(
                current_params=params,
                target_params=None,
                blend_alpha=0.0,
                speaking_level=0.0,
                listening=False,
                delta_time=self._last_delta_time
            )
            return
        frame = self.renderer.get_emotion_frame(
            self.state.current_emotion, speaking=False
        )
        if frame:
            self.renderer.render_frame(frame)
def _check_gpio_exit(self) -> bool:
    """
    Check if GPIO exit button is pressed
    Returns:
        True if button pressed, False otherwise
    """
    if not GPIO_AVAILABLE or not self.gpio_enabled:
        return False
    try:
        return not GPIO.input(self.gpio_exit_pin)
    except (RuntimeError, ValueError, OSError) as e:
        logger.error("GPIO read error: %s", e)
        return False
def set_emotion(self, emotion: str, transition_duration: float = 0.5):
    """
    Set display emotion with smooth transition
    Args:
        emotion: Emotion name (e.g., 'happy', 'sad', 'excited')
        transition_duration: Transition duration in seconds
    """
    self.command_queue.put({
        'type': 'SET_EMOTION',
        'emotion': emotion,
        'duration': transition_duration
    })
def set_listening(self, active: bool):
    """
    Set listening state
    Args:
        active: True to show listening animation, False to return
    """
```

```
        self.command_queue.put({
            'type': 'SET_LISTENING',
            'active': active
        })
    def set_speaking(self, active: bool, level: Optional[float] = None):
        """
        Set speaking state
        Args:
            active: True to enable speaking animation, False to stop
            level: Optional mouth intensity (0-1) for procedural mode
        """
        self.command_queue.put({
            'type': 'SET_SPEAKING',
            'active': active,
            'level': level
        })
    def set_effect_callback(self, callback: Callable[[Dict], None]):
        """
        Set an optional effect callback invoked on gesture effects.
        Args:
            callback: Callable receiving a dict with effect info.
        """
        self.effect_callback = callback
    def set_exit_callback(self, callback: Callable[[], None]):
        """
        Set a callback invoked when the GPIO exit button is pressed.
        """
        self.exit_callback = callback
    def cleanup(self):
        """Clean up display resources and GPIO"""
        logger.info("Cleaning up emotion display...")
        self.stop()
        self.renderer.cleanup()
        if GPIO_AVAILABLE and self.gpio_enabled:
            try:
                GPIO.cleanup()
                logger.info("GPIO cleanup complete")
            except (RuntimeError, ValueError, OSError) as e:
                logger.error("GPIO cleanup error: %s", e)
        logger.info("EmotionDisplay cleanup complete")
    def _build_emotion_params(self) -> Dict[str, Dict]:
        """
        Build emotion parameter presets with optional config overrides.
        """
        presets = {
            key: val.copy() for key, val in DEFAULT_EMOTION_PRESETS.items()
        }
        overrides = self.procedural_config.get('presets', {})
        for key, override in overrides.items():
            if key in presets and isinstance(override, dict):
                merged = presets[key].copy()
                merged.update(override)
                presets[key] = merged
            elif isinstance(override, dict):
                presets[key] = override
        return presets
    def _get_emotion_params(self, emotion: str) -> Dict:
        if emotion in self.emotion_params:
            return self.emotion_params[emotion]
        if "happy" in self.emotion_params:
            return self.emotion_params["happy"]
        return list(self.emotion_params.values())[0]
    def _handle_touch_event(self, event):
        """
        Handle raw pygame touch/mouse events and detect gestures.
        """
        if pygame is None:
            return
        if self.state.petting_active:
            return
        if event.type == pygame.MOUSEBUTTONDOWN:
            self._touch_start_pos = event.pos
            self._touch_down_pos = event.pos
            self._touch_down_time = time.time()
            self._drag_distance = 0.0
        elif event.type == pygame.MOUSEMOTION and event.buttons[0]:
            if self._touch_down_pos:
                dx = event.pos[0] - self._touch_down_pos[0]
                dy = event.pos[1] - self._touch_down_pos[1]
                self._drag_distance += abs(dx) + abs(dy)
                self._touch_down_pos = event.pos
        elif event.type == pygame.MOUSEBUTTONUP:
            if self._touch_start_pos is None:
                return
            up_pos = event.pos
            start_pos = self._touch_start_pos
            duration = time.time() - self._touch_down_time
            dist = self._drag_distance
            dx = up_pos[0] - start_pos[0]
            dy = up_pos[1] - start_pos[1]
            self._touch_start_pos = None
            self._touch_down_pos = None
            self._drag_distance = 0.0
            gesture = self._classify_gesture(duration, dist, dx, dy)
            if gesture:
                self._trigger_gesture_effect(gesture)
    def _discard_touch_events(self):
        """
        Drop queued touch/mouse events so they don't fire after a petting lock.
        """
        if pygame is None:
            return
        touch_events = (
            pygame.MOUSEBUTTONDOWN,
            pygame.MOUSEBUTTONUP,
            pygame.MOUSEMOTION,
        )
        pygame.event.clear(touch_events)
    def _classify_gesture(
        self, duration: float, dist: float, dx: float, dy: float
    ) -> Optional[str]:
        """
        Classify gesture based on simple thresholds.
        """
        tap_dist = float(self.touch_thresholds.get('tap_distance', 20))
        double_tap_window = float(
            self.touch_thresholds.get('double_tap_window', 0.35)
        )
        long_press_time = float(self.touch_thresholds.get('long_press', 0.6))
        drag_dist = float(self.touch_thresholds.get('drag_distance', 60))
        circle_dist = float(self.touch_thresholds.get('circle_distance', 140))
        circle_return = float(self.touch_thresholds.get('circle_return', 45))
        now = time.time()
        closure = abs(dx) + abs(dy)
        closure_ratio = closure / max(dist, 1e-6)
        if duration >= long_press_time and dist < drag_dist:
            return "long_press"
        if dist < tap_dist and duration < long_press_time:
            if (
                self.pending_tap_time > 0.0
                and now - self.pending_tap_time <= double_tap_window
            ):
                self.pending_tap_time = 0.0
                return "double_tap"
            self.pending_tap_time = now
            return None
        if (
            dist >= circle_dist
            and closure <= circle_return
            and closure_ratio <= 0.25
        ):
            return "scroll"
        if dist >= drag_dist:
            return "drag"
        return None
    def _trigger_gesture_effect(self, gesture: str):
        """
        Trigger configured effect for a gesture with rate limiting.
        """
        now = time.time()
        if now - self.state.last_gesture_time < self.gesture_cooldown:
            return
        if now < self.state.gesture_busy_until:
            return
        if self.state.petting_active:
            return
        self.state.last_gesture_time = now
        effect = self.gesture_effects.get(gesture)
        if not effect:
            logger.debug("Gesture %s has no configured effect", gesture)
            return
        self.state.petting_active = True
        self.state.gesture_busy_until = now + self.effect_busy_window
        self._discard_touch_events()
        if now - self._last_effect_time < self.effect_queue_cooldown:
            self.state.petting_active = False
            logger.debug("Effect suppressed by cooldown")
            return
        self._last_effect_time = now
        emotion = effect.get('emotion')
        if emotion:
            self.set_emotion(emotion, transition_duration=0.4)
        busy_seconds = self.effect_busy_window
        speak_text = effect.get('speak')
        if isinstance(speak_text, str) and speak_text.strip():
            estimated = (len(speak_text) / 10.0) + 1.5
            busy_seconds = max(busy_seconds, estimated)
        self.state.gesture_busy_until = now + busy_seconds
        if self.effect_callback:
            def _cb_wrapper():
                try:
                    self.effect_callback(effect)
                except (
                    RuntimeError,
                    ValueError,
                    OSError,
                    TypeError,
                ) as exc:
                    logger.error("Effect callback failed: %s", exc)
            threading.Thread(target=_cb_wrapper, daemon=True).start()
        else:
            self.command_queue.put({
```

```
            'type': 'APPLY_EFFECT',
            'effect': effect
        })
def _apply_effect(self, effect: Dict):
    """
    Handle non-emotion effects internally (placeholder for future hooks).
    """
    sound = effect.get('sound')
    speak = effect.get('speak')
    hardware = effect.get('hardware')
    if sound:
        logger.info("Gesture sound requested: %s", sound)
    if speak:
        logger.info("Gesture speak requested: %s", speak)
    if hardware:
        logger.info("Gesture hardware action: %s", hardware)
```

### src/expression/procedural_face.py

```
"""
Procedural face renderer for lightweight, animatable expressions.
"""
import math
import random
from typing import Dict, Tuple, Optional
import pygame
def _lerp(a: float, b: float, t: float) -> float:
    return a + (b - a) * max(0.0, min(1.0, t))
def _clamp01(x: float) -> float:
    return max(0.0, min(1.0, x))
class ProceduralFaceRenderer:
    """
    Draws a simple face (eyes, brows, mouth) using Pygame primitives so we can
    animate without swapping PNGs. All colors and geometry are driven by the
    supplied emotion parameters and blended when transitioning.
    """
    def __init__(self, screen_size: Tuple[int, int], config: Dict):
        self.screen_size = screen_size
        self.surface = pygame.Surface(screen_size, pygame.SRCALPHA)
        blink_range = config.get("blink_interval", [3.0, 6.0])
        self.blink_interval_range = (
            float(blink_range[0]),
            float(blink_range[1]) if len(blink_range) > 1 else float(blink_range[0]),
        )
        self.blink_duration = float(config.get("blink_duration", 0.12))
        self.eye_jitter = float(config.get("eye_jitter", 1.5))
        self.mouth_smooth = float(config.get("mouth_smooth", 8.0))
        self.transition_smooth = float(config.get("transition_smooth", 6.0))
        self.listening_pulse_speed = float(config.get("listening_pulse_speed", 1.5))
        self.listening_pulse_strength = float(config.get("listening_pulse_strength", 0.08))
        self.listening_glow_color = tuple(config.get("listening_glow_color", [0, 200, 255]))
        self.listening_glow_alpha = float(config.get("listening_glow_alpha", 0.35))
        self.listening_glow_thickness = int(config.get("listening_glow_thickness", 6))
        self.background_color = tuple(config.get("background", [0, 0, 0]))
        self._blink_timer = 0.0
        self._time_since_blink = 0.0
        self._next_blink = self._random_blink_interval()
        self._is_blinking = False
        self._mouth_level = 0.0
        self._speaking_target = 0.0
        self._listening_phase = 0.0
    def _random_blink_interval(self) -> float:
        return random.uniform(*self.blink_interval_range)
    def update_state(self, delta_time: float, speaking_level: float, listening: bool):
        self._speaking_target = _clamp01(speaking_level)
        smooth = max(1e-3, self.mouth_smooth)
        blend = 1.0 - math.exp(-smooth * max(delta_time, 0.0))
        self._mouth_level = _lerp(self._mouth_level, self._speaking_target, blend)
        self._time_since_blink += delta_time
        if not self._is_blinking and self._time_since_blink >= self._next_blink:
            self._is_blinking = True
            self._blink_timer = 0.0
        if self._is_blinking:
            self._blink_timer += delta_time
            if self._blink_timer >= self.blink_duration:
                self._is_blinking = False
                self._time_since_blink = 0.0
                self._next_blink = self._random_blink_interval()
        if listening:
            self._listening_phase += delta_time * self.listening_pulse_speed
        else:
            self._listening_phase = 0.0
    def render(
        self,
        current_params: Dict,
        target_params: Optional[Dict],
        blend_alpha: float,
        listening: bool,
    ) -> pygame.Surface:
        params = self._blend_params(current_params, target_params, blend_alpha)
        w, h = self.screen_size
        self.surface.fill(self.background_color)
        eye_spacing = params.get("eye_spacing", 90)
        eye_width = params.get("eye_width", 42)
        eye_height = params.get("eye_height", 42)
        eye_y = params.get("eye_y", int(h * 0.42))
        pupil_size = params.get("pupil_size", 12)
        brow_raise = params.get("brow_raise", 0.0)
        brow_slant = params.get("brow_slant", 0.0)
        mouth_width = params.get("mouth_width", 120)
        mouth_height = params.get("mouth_height", 24)
        mouth_curve = params.get("mouth_curve", 0.0)
        mouth_base = params.get("mouth_open", 0.05)
        eye_color = tuple(params.get("eye_color", [240, 240, 240]))
        pupil_color = tuple(params.get("pupil_color", [20, 20, 20]))
        brow_color = tuple(params.get("brow_color", [220, 220, 220]))
        mouth_color = tuple(params.get("mouth_color", [240, 120, 120]))
        listening_scale = 1.0 + (math.sin(self._listening_phase) * self.listening_pulse_strength if
listening else 0.0)
        for direction in (-1, 1):
            jitter_x = random.uniform(-self.eye_jitter, self.eye_jitter)
            jitter_y = random.uniform(-self.eye_jitter, self.eye_jitter)
            eye_center = (
                int(w / 2 + direction * eye_spacing * listening_scale + jitter_x),
                int(eye_y * listening_scale + jitter_y),
            )
            blink_scale = 0.2 if self._is_blinking else 1.0
            self._draw_eye(eye_center, int(eye_width * listening_scale), int(eye_height * listening_
scale * blink_scale),
                           eye_color, pupil_color, pupil_size, params.get("pupil_offset", 0))
            self._draw_brow(eye_center, eye_width, eye_height, brow_raise, brow_slant * direction, b
row_color)
        mouth_open = mouth_base + self._mouth_level * params.get("mouth_sensitivity", 0.6)
        mouth_open = _clamp01(mouth_open)
        self._draw_mouth((w // 2, int(h * 0.7)), mouth_width, mouth_height, mouth_curve, mouth_open,
 mouth_color)
        if listening and self.listening_glow_alpha > 0.0:
            pulse = 0.5 + 0.5 * math.sin(self._listening_phase)
            alpha = int(255 * self.listening_glow_alpha * pulse)
            if alpha > 0:
                overlay = pygame.Surface(self.screen_size, pygame.SRCALPHA)
                color = (
                    self.listening_glow_color[0],
                    self.listening_glow_color[1],
                    self.listening_glow_color[2],
                    alpha,
                )
                thickness = max(2, self.listening_glow_thickness)
                rect = overlay.get_rect().inflate(-thickness, -thickness)
                pygame.draw.rect(overlay, color, rect, thickness)
                self.surface.blit(overlay, (0, 0))
        return self.surface
    def _blend_params(self, current: Dict, target: Optional[Dict], alpha: float) -> Dict:
        if not target:
            return current
        t = _clamp01(alpha)
        blended = {}
        keys = set(current.keys()) | set(target.keys())
        for key in keys:
            a = current.get(key, 0.0)
            b = target.get(key, a)
            if isinstance(a, (list, tuple)) and isinstance(b, (list, tuple)) and len(a) == len(b):
                blended[key] = [ _lerp(x, y, t) for x, y in zip(a, b) ]
            elif isinstance(a, (int, float)) and isinstance(b, (int, float)):
                blended[key] = _lerp(float(a), float(b), t)
            else:
                blended[key] = b
        return blended
    def _draw_eye(
        self,
        center: Tuple[int, int],
        width: int,
        height: int,
        eye_color: Tuple[int, int, int],
        pupil_color: Tuple[int, int, int],
        pupil_size: float,
        pupil_offset: float,
    ):
        rect = pygame.Rect(0, 0, width, height)
        rect.center = center
        pygame.draw.ellipse(self.surface, eye_color, rect)
        pupil_rect = pygame.Rect(0, 0, int(pupil_size), int(pupil_size))
        pupil_rect.center = (center[0] + pupil_offset, center[1])
        pygame.draw.circle(self.surface, pupil_color, pupil_rect.center, pupil_rect.width // 2)
    def _draw_brow(
        self,
        eye_center: Tuple[int, int],
        eye_width: int,
        eye_height: int,
        raise_amt: float,
        slant: float,
        color: Tuple[int, int, int],
    ):
        start = (
            int(eye_center[0] - eye_width * 0.6),
            int(eye_center[1] - eye_height * (0.8 + raise_amt) - slant * 10),
        )
        end = (
            int(eye_center[0] + eye_width * 0.6),
            int(eye_center[1] - eye_height * (0.8 + raise_amt) + slant * 10),
        )
        pygame.draw.line(self.surface, color, start, end, 4)
    def _draw_mouth(
        self,
        center: Tuple[int, int],
        width: int,
        height: int,
        curve: float,
        openness: float,
        color: Tuple[int, int, int],
    ):
        w2 = width // 2
        h2 = max(2, int(height * (0.3 + openness)))
        curve_offset = int(curve * height)
        start = (center[0] - w2, center[1])
        end = (center[0] + w2, center[1])
        control = (center[0], center[1] + curve_offset)
        points = []
        steps = 20
```

```
for i in range(steps + 1):
    t = i / steps
    x = int((1 - t) * (1 - t) * start[0] + 2 * (1 - t) * t * control[0] + t * t * end[0])
    y = int((1 - t) * (1 - t) * start[1] + 2 * (1 - t) * t * control[1] + t * t * end[1])
    points.append((x, y))
pygame.draw.lines(self.surface, color, False, points, h2)
```

### src/expression/transition_controller.py

```
"""
Transition Controller for Emotion Expression Pipeline
Manages smooth cross-fade transitions between emotions
"""
import time
import logging
from typing import Optional, Tuple
logger = logging.getLogger(__name__)
class TransitionController:
    """
    Manages smooth transitions between emotions using alpha blending
    Tracks transition progress over time for interpolation
    """
    def __init__(self):
        """Initialize transition controller"""
        self.is_active = False
        self.from_emotion: Optional[str] = None
        self.to_emotion: Optional[str] = None
        self.duration: float = 0.5
        self.start_time: float = 0.0
        self.elapsed_time: float = 0.0
        self.alpha: float = 0.0
    def start_transition(self, from_emotion: str, to_emotion: str, duration: float = 0.5):
        """
        Start a new emotion transition
        Args:
            from_emotion: Current emotion to transition from
            to_emotion: Target emotion to transition to
            duration: Transition duration in seconds
        """
        self.from_emotion = from_emotion
        self.to_emotion = to_emotion
        self.duration = max(0.1, duration)
        self.start_time = time.time()
        self.elapsed_time = 0.0
        self.alpha = 0.0
        self.is_active = True
        logger.debug(f"Transition started: {from_emotion} ? {to_emotion} ({duration}s)")
    def update(self, delta_time: float) -> Tuple[str, str, float]:
        """
        Update transition progress
        Args:
            delta_time: Time elapsed since last update (seconds)
        Returns:
            Tuple of (from_emotion, to_emotion, alpha)
            alpha: 0.0 = show from_emotion, 1.0 = show to_emotion
        """
        if not self.is_active:
            return (self.from_emotion or 'happy', self.to_emotion or 'happy', 1.0)
        self.elapsed_time += delta_time
        self.alpha = min(1.0, self.elapsed_time / self.duration)
        if self.alpha >= 1.0:
            self.is_active = False
            self.from_emotion = self.to_emotion
            logger.debug(f"Transition complete: now showing {self.to_emotion}")
        return (self.from_emotion, self.to_emotion, self.alpha)
    def is_transitioning(self) -> bool:
        """
        Check if transition is currently in progress
        Returns:
            True if transitioning, False otherwise
        """
        return self.is_active
    def skip_to_end(self):
        """Skip to end of transition instantly"""
        if self.is_active:
            self.alpha = 1.0
            self.is_active = False
            self.from_emotion = self.to_emotion
            logger.debug(f"Transition skipped to end: {self.to_emotion}")
    def get_current_emotion(self) -> str:
        """
        Get the current emotion (from or to depending on transition state)
        Returns:
            Current emotion name
        """
        if self.is_active and self.alpha >= 0.5:
            return self.to_emotion or 'happy'
        else:
            return self.from_emotion or 'happy'
    def get_progress(self) -> float:
        """
        Get transition progress
        Returns:
            Progress from 0.0 to 1.0
        """
        return self.alpha
```

```
### src/llm/__init__.py

"""
LLM Integration Module
Complete conversational AI pipeline with:
- Speech-to-Text (Whisper)
- LLM Integration (Ollama)
- Text-to-Speech (pyttsx3)
- Conversation Management
- Full Voice Pipeline
"""
from .stt_engine import STTEngine, RealtimeSTT
from .voice_pipeline import VoicePipeline
from .ollama_client import OllamaClient
from .tts_engine import TTSEngine
from .conversation_manager import ConversationManager
from .conversation_pipeline import ConversationPipeline
__all__ = [
    'STTEngine',
    'RealtimeSTT',
    'VoicePipeline',
    'OllamaClient',
    'TTSEngine',
    'ConversationManager',
    'ConversationPipeline',
]
```

### src/llm/conversation_manager.py

```
"""
Conversation Manager
Manages conversation context, personality integration, and LLM orchestration
"""
import logging
import time
import sys
import re
from pathlib import Path
from typing import Dict, List, Optional, Tuple, Iterator
from collections import deque
sys.path.insert(0, str(Path(__file__).parent.parent))
from llm.ollama_client import OllamaClient
logger = logging.getLogger(__name__)
class StreamingEmotionParser:
    """
    Parses emotion tags from streaming LLM token stream
    Emits (emotion, text) segments when complete sentences are detected
    """
    def __init__(self, valid_emotions: set, segment_timeout: float = 2.0, min_segment_length: int =
5):
        """
        Initialize streaming emotion parser
        Args:
            valid_emotions: Set of valid emotion strings
            segment_timeout: Seconds before forcing segment emit without sentence boundary
            min_segment_length: Minimum characters before emitting a segment
        """
        self.valid_emotions = valid_emotions
        self.segment_timeout = segment_timeout
        self.min_segment_length = min_segment_length
        self.reset()
    def reset(self):
        """Reset parser state"""
        self.state = "ACCUMULATING"
        self.buffer = ""
        self.current_emotion = None
        self.current_text = ""
        self.last_emit_time = time.time()
        logger.debug("StreamingEmotionParser reset")
    def add_token(self, token: str) -> List[Tuple[str, str]]:
        """
        Add streaming token to parser
        Args:
            token: Single token from LLM stream
        Returns:
            List of (emotion, text) tuples ready to be spoken (may be empty)
        """
        segments = []
        self.buffer += token
        if self.state == "ACCUMULATING":
            tag_match = re.search(r'\[(\w+)\]', self.buffer)
            if tag_match:
                emotion = tag_match.group(1).lower()
                if emotion in self.valid_emotions:
                    self.current_emotion = emotion
                    self.state = "TAG_FOUND"
                    text_after_tag = self.buffer[tag_match.end():].lstrip()
                    self.current_text = text_after_tag
                    self.buffer = ""
                    self.last_emit_time = time.time()
                    logger.debug(f"Emotion tag found: [{emotion}]")
                else:
                    logger.warning(f"Invalid emotion '{emotion}', using 'happy'")
                    self.current_emotion = 'happy'
                    self.state = "TAG_FOUND"
                    text_after_tag = self.buffer[tag_match.end():].lstrip()
                    self.current_text = text_after_tag
                    self.buffer = ""
                    self.last_emit_time = time.time()
        elif self.state == "TAG_FOUND":
            self.current_text += token
            if self._is_segment_boundary():
                segment_text = self._prepare_segment_text()
                if segment_text:
                    segments.append((self.current_emotion, segment_text))
                    logger.debug(f"Segment ready: ({self.current_emotion}) {segment_text[:30]}...")
                self.state = "ACCUMULATING"
                self.buffer = ""
                self.current_text = ""
                self.current_emotion = None
                self.last_emit_time = time.time()
        return segments
    def _is_segment_boundary(self) -> bool:
        """
        Determine if current text forms a complete segment ready to emit
        Returns:
            True if segment is ready
        """
        if re.search(r'[.!?]\s*$', self.current_text):
            return True
        time_since_last_emit = time.time() - self.last_emit_time
        if time_since_last_emit > self.segment_timeout and len(self.current_text) >= self.min_segmen
t_length:
            logger.debug(f"Segment timeout reached ({time_since_last_emit:.1f}s), forcing emit")
            return True
        if '[' in self.current_text:
            parts = self.current_text.split('[')
            if len(parts) > 1 and len(parts[-1]) < 3:
                self.current_text = '['.join(parts[:-1])
                if len(self.current_text) >= self.min_segment_length:
                    logger.debug("New emotion tag detected, emitting previous segment")
                    self.buffer = '[' + parts[-1]
                    return True
        return False
    def _prepare_segment_text(self) -> str:
        """
        Prepare segment text for emission (clean up formatting)
        Returns:
            Cleaned segment text
        """
        text = self.current_text.strip()
        if text.endswith('['):
            text = text[:-1].strip()
        return text
    def flush(self) -> Optional[Tuple[str, str]]:
        """
        Emit any remaining buffered content at end of stream
        Returns:
            (emotion, text) tuple if content available, None otherwise
        """
        if self.state == "TAG_FOUND" and self.current_text.strip():
            emotion = self.current_emotion or 'happy'
            text = self._prepare_segment_text()
            logger.debug(f"Flushing final segment: ({emotion}) {text[:30]}...")
            return (emotion, text)
        elif self.buffer.strip():
            logger.debug(f"Flushing buffer with default emotion: {self.buffer[:30]}...")
            return ('happy', self.buffer.strip())
        return None
class ConversationManager:
    """
    Manages conversation flow with context, personality, and emotion awareness
    """
    def __init__(
        self,
        config: dict,
        emotion_engine=None,
        user_memory=None,
        conversation_history=None
    ):
        """
        Initialize conversation manager
        Args:
            config: Configuration dictionary
            emotion_engine: Optional EmotionEngine instance
            user_memory: Optional UserMemory instance
            conversation_history: Optional ConversationHistory instance
        """
        self.config = config
        self.emotion_engine = emotion_engine
        self.user_memory = user_memory
        self.conversation_history_db = conversation_history
        self.llm = OllamaClient(config)
        self.context_window = config.get('memory', {}).get('conversation', {}).get('context_window',
 10)
        self.max_history = config.get('memory', {}).get('conversation', {}).get('max_history', 50)
        self.conversation_history: deque = deque(maxlen=self.max_history)
        self.current_context: deque = deque(maxlen=self.context_window)
        self.current_user_id: Optional[int] = None
        self.current_user_name: str = "friend"
        self.conversation_start_time = time.time()
        self.message_count = 0
        self.session_id = self._generate_session_id()
        self.max_response_length = 240
        self.response_filters = [
            self._ensure_short,
            self._ensure_pet_like,
            self._add_expressiveness
        ]
        self.valid_emotions = {
            'happy', 'sad', 'excited', 'curious', 'sleepy',
            'lonely', 'playful', 'scared', 'angry', 'loving',
            'bored', 'surprised'
        }
        logger.info("Conversation manager initialized")
    def process_user_input(
        self,
        user_text: str,
        user_id: Optional[int] = None
    ) -> Tuple[str, Dict]:
        """
        Process user input and generate response
        Args:
            user_text: User's message
            user_id: Optional user ID
        Returns:
            Tuple of (response_text, metadata)
        """
        if not user_text or not user_text.strip():
            logger.warning("Empty user input")
            return "...", {'error': 'empty_input'}
        if user_id is not None:
            self.current_user_id = user_id
            self._update_user_name()
        start_time = time.time()
        formatted_context = self._format_context_for_llm()
        result = self.llm.generate_with_personality(
            user_input=user_text,
            user_name=self.current_user_name,
            context=formatted_context
        )
        response_time = time.time() - start_time
        raw_response = result.get('response', '')
        emotion_segments = self._parse_emotion_segments(raw_response)
        final_emotion = emotion_segments[-1][0] if emotion_segments else 'happy'
        filtered_segments = []
        for emotion, text in emotion_segments:
            filtered_text = self._filter_response(text)
            filtered_segments.append((emotion, filtered_text))
        filtered_response = ' '.join(text for _, text in filtered_segments)
        self._add_to_history('user', user_text)
        self._add_to_history(
            'assistant',
            filtered_response,
            emotion=final_emotion,
            tokens=result.get('tokens', 0)
        )
        self._update_context(user_text, filtered_response)
        if self.emotion_engine:
            try:
                emotion_sequence = [emotion for emotion, _ in emotion_segments]
                if hasattr(self.emotion_engine, 'process_emotion_sequence'):
                    self.emotion_engine.process_emotion_sequence(emotion_sequence)
                else:
                    self.emotion_engine.set_emotion_from_llm(final_emotion)
            except Exception as e:
                logger.warning(f"Error updating emotion engine: {e}")
                if hasattr(self.emotion_engine, 'on_voice_interaction'):
                    self.emotion_engine.on_voice_interaction()
        self.message_count += 1
        current_energy = self._get_current_energy()
        metadata = {
            'emotion': final_emotion,
            'emotion_segments': filtered_segments,
            'energy': current_energy,
            'response_time': response_time,
            'tokens': result.get('tokens', 0),
            'model': result.get('model', 'unknown'),
            'message_count': self.message_count,
            'fallback': result.get('fallback', False)
        }
        logger.info(f"Response generated in {response_time:.2f}s ({metadata['tokens']} tokens, {len(
emotion_segments)} emotion segment(s), final: {final_emotion})")
        return filtered_response, metadata
    def stream_generate_with_personality(
        self,
        user_text: str,
        user_id: Optional[int] = None
    ) -> Iterator[Tuple[str, str]]:
        """
        Stream response generation with emotion parsing
        Yields (emotion, text) tuples as segments become ready
        This enables faster perceived response time by speaking as soon as
        the first complete sentence is generated, rather than waiting for
        the entire response.
        Args:
            user_text: User's message
            user_id: Optional user ID
        Yields:
            (emotion, text) tuples for each complete segment
        """
        if not user_text or not user_text.strip():
            logger.warning("Empty user input for streaming")
            yield ('happy', '...')
            return
        if user_id is not None:
            self.current_user_id = user_id
            self._update_user_name()
        system_prompt = self.llm.personality_template.format(
            user_name=self.current_user_name
        )
        streaming_config = self.config.get('llm', {}).get('streaming', {})
        segment_timeout = streaming_config.get('segment_timeout', 2.0)
        min_segment_length = streaming_config.get('min_segment_length', 5)
        parser = StreamingEmotionParser(
            self.valid_emotions,
            segment_timeout=segment_timeout,
            min_segment_length=min_segment_length
        )
        logger.info("Starting streaming generation...")
        formatted_context = self._format_context_for_llm()
        all_segments = []
        segment_count = 0
        try:
            for token in self.llm.stream_generate(user_text, system_prompt=system_prompt, context=fo
rmatted_context):
                segments = parser.add_token(token)
                for emotion, text in segments:
                    filtered_text = self._filter_response(text)
                    if filtered_text:
                        segment_count += 1
                        all_segments.append((emotion, filtered_text))
                        logger.info(f"Streaming segment {segment_count}: ({emotion}) {filtered_text[
:40]}...")
                        yield (emotion, filtered_text)
            final_segment = parser.flush()
            if final_segment:
                emotion, text = final_segment
                filtered_text = self._filter_response(text)
                if filtered_text:
                    segment_count += 1
                    all_segments.append((emotion, filtered_text))
                    logger.info(f"Streaming final segment: ({emotion}) {filtered_text[:40]}...")
                    yield (emotion, filtered_text)
            combined_text = ' '.join(text for _, text in all_segments)
            self._add_to_history('user', user_text)
            self._add_to_history('assistant', combined_text)
            self._update_context(user_text, combined_text)
            if self.emotion_engine and all_segments:
                try:
                    emotion_sequence = [emotion for emotion, _ in all_segments]
                    if hasattr(self.emotion_engine, 'process_emotion_sequence'):
                        self.emotion_engine.process_emotion_sequence(emotion_sequence)
                    else:
                        final_emotion = all_segments[-1][0]
                        self.emotion_engine.set_emotion_from_llm(final_emotion)
                except Exception as e:
```

```
                logger.warning(f"Error updating emotion engine: {e}")
            self.message_count += 1
            logger.info(f"Streaming complete: {segment_count} segments generated")
        except Exception as e:
            logger.error(f"Error in streaming generation: {e}", exc_info=True)
            yield ('happy', "Sorry, I had trouble with that.")
    def _build_context(self) -> List[str]:
        """
        Build conversation context for LLM
        Returns:
            List of formatted context messages
        """
        context = []
        for role, message in self.current_context:
            if role == 'user':
                context.append(f"User: {message}")
            else:
                context.append(f"Assistant: {message}")
        return context
    def _update_context(self, user_msg: str, assistant_msg: str):
        """
        Update the sliding context window
        Args:
            user_msg: User's message
            assistant_msg: Assistant's response
        """
        self.current_context.append(('user', user_msg))
        self.current_context.append(('assistant', assistant_msg))
    def _add_to_history(self, role: str, message: str, emotion: str = None, tokens: int = 0):
        """
        Add message to full conversation history (in-memory and database)
        Args:
            role: 'user' or 'assistant'
            message: Message content
            emotion: Optional emotion (for assistant messages)
            tokens: Optional token count (for assistant messages)
        """
        self.conversation_history.append({
            'role': role,
            'message': message,
            'emotion': emotion,
            'tokens': tokens,
            'timestamp': time.time()
        })
        if self.conversation_history_db:
            try:
                self.conversation_history_db.save_message(
                    user_id=self.current_user_id,
                    session_id=self.session_id,
                    role=role,
                    message=message,
                    emotion=emotion,
                    tokens=tokens
                )
            except Exception as e:
                logger.warning(f"Failed to save message to database: {e}")
    def _get_current_emotion(self) -> str:
        """
        Get current emotion from emotion engine
        Returns:
            Emotion state string
        """
        if self.emotion_engine:
            try:
                return self.emotion_engine.get_emotion()
            except Exception as e:
                logger.error(f"Error getting emotion: {e}")
        return "happy"
    def _get_current_energy(self) -> float:
        """
        Get current energy level
        Returns:
            Energy level (0-1)
        """
        if self.emotion_engine:
            try:
                return self.emotion_engine.energy_level
            except Exception as e:
                logger.error(f"Error getting energy: {e}")
        return 0.7
    def _format_context_for_llm(self) -> Optional[List[str]]:
        """
        Format conversation context for LLM
        Returns:
            Formatted context as list of strings, or None if empty
        """
        if not self.current_context:
            return None
        formatted = []
        for role, message in self.current_context:
            if role == 'user':
                formatted.append(f"User: {message}")
            else:
                formatted.append(f"Assistant: {message}")
        return formatted
    def _update_user_name(self):
        """Update current user name from memory"""
        if self.user_memory and self.current_user_id:
            try:
                user_profile = self.user_memory.get_user_by_id(self.current_user_id)
                if user_profile:
                    self.current_user_name = user_profile['name']
                    logger.info(f"Loaded user profile: {self.current_user_name} (ID: {self.current_u
ser_id})")
                else:
                    self.current_user_name = "friend"
            except Exception as e:
                logger.error(f"Error getting user name: {e}")
                self.current_user_name = "friend"
    def _generate_session_id(self) -> str:
        """
        Generate unique session ID
        Returns:
            Session ID string
        """
        import uuid
        return str(uuid.uuid4())
    def _parse_emotion_segments(self, response: str) -> List[Tuple[str, str]]:
        """
        Parse all emotion tags from LLM response and split into segments
        Supports multiple emotions throughout the response:
        "[excited] Hello! [curious] What's that?"
        ? [("excited", "Hello!"), ("curious", "What's that?")]
        Args:
            response: Raw LLM response with emotion tag(s)
        Returns:
            List of (emotion, text) tuples for each segment
        """
        response = response.strip()
        pattern = r'\[(\w+)\]\s*([^\[]+)'
        matches = re.findall(pattern, response)
        if not matches:
            logger.warning("No emotion tags found in response, using default 'happy'")
            return [('happy', response)]
        segments = []
        for emotion_raw, text in matches:
            emotion = emotion_raw.lower().strip()
            text = text.strip()
            if not text:
                continue
            if emotion not in self.valid_emotions:
                logger.warning(f"Invalid emotion '{emotion}', using 'happy' for this segment")
                emotion = 'happy'
            segments.append((emotion, text))
        if not segments:
            logger.warning("All emotion segments were empty, using default")
            return [('happy', response)]
        logger.debug(f"Parsed {len(segments)} emotion segment(s)")
        return segments
    def _parse_emotion(self, response: str) -> Tuple[str, str]:
        """
        Legacy method for backward compatibility
        Parses first emotion and returns combined message
        Args:
            response: Raw LLM response with emotion tag(s)
        Returns:
            Tuple of (first_emotion, message_without_tags)
        """
        segments = self._parse_emotion_segments(response)
        if not segments:
            return 'happy', response
        first_emotion = segments[0][0]
        combined_text = ' '.join(text for _, text in segments)
        return first_emotion, combined_text
    def _filter_response(self, response: str) -> str:
        """
        Apply filters to ensure response is appropriate
        Args:
            response: Raw LLM response
        Returns:
            Filtered response
        """
        filtered = response
        for filter_func in self.response_filters:
            try:
                filtered = filter_func(filtered)
            except Exception as e:
                logger.error(f"Filter error: {e}")
        return filtered
    def _ensure_short(self, response: str) -> str:
        """
        Ensure response is short (pet-like)
        Args:
            response: Input response
        Returns:
            Shortened response
        """
        if len(response) > self.max_response_length:
            sentences = response.split('. ')
            if sentences:
                short = '. '.join(sentences[:2])
                if not short.endswith('.'):
```

```
                short += '.'
            return short
        else:
            return response[:self.max_response_length] + '...'
    return response
def _ensure_pet_like(self, response: str) -> str:
    """
    Ensure response sounds pet-like
    Args:
        response: Input response
    Returns:
        Pet-like response
    """
    replacements = {
        'I apologize': 'Sorry!',
        'I understand': 'I get it!',
        'However': 'But',
        'Nevertheless': 'But',
        'Furthermore': 'Also',
    }
    result = response
    for formal, casual in replacements.items():
        result = result.replace(formal, casual)
    return result
def _add_expressiveness(self, response: str) -> str:
    """
    Add expressiveness to response (optional)
    Args:
        response: Input response
    Returns:
        More expressive response
    """
    return response
def get_conversation_summary(self) -> str:
    """
    Get summary of conversation
    Returns:
        Summary string
    """
    duration = time.time() - self.conversation_start_time
    duration_min = duration / 60.0
    summary = f"Conversation Summary:\n"
    summary += f"  Duration: {duration_min:.1f} minutes\n"
    summary += f"  Messages: {self.message_count}\n"
    summary += f"  Current emotion: {self._get_current_emotion()}\n"
    summary += f"  User: {self.current_user_name}\n"
    return summary
def get_conversation_history(self, limit: Optional[int] = None) -> List[Dict]:
    """
    Get conversation history
    Args:
        limit: Optional limit on number of messages
    Returns:
        List of message dictionaries
    """
    history = list(self.conversation_history)
    if limit:
        history = history[-limit:]
    return history
def clear_context(self):
    """Clear conversation context (but keep history)"""
    self.current_context.clear()
    logger.info("Conversation context cleared")
def clear_history(self):
        """Clear all conversation history"""
        self.conversation_history.clear()
        self.current_context.clear()
        self.message_count = 0
        self.conversation_start_time = time.time()
        logger.info("Conversation history cleared")
    def save_conversation(self, filename: str):
        """
        Save conversation history to file
        Args:
            filename: Output file path
        """
        import json
        try:
            data = {
                'user_name': self.current_user_name,
                'start_time': self.conversation_start_time,
                'message_count': self.message_count,
                'history': list(self.conversation_history)
            }
            with open(filename, 'w') as f:
                json.dump(data, f, indent=2)
            logger.info(f"Conversation saved to {filename}")
        except Exception as e:
            logger.error(f"Error saving conversation: {e}")
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    import yaml
    config_path = Path(__file__).parent.parent.parent / 'config' / 'settings.yaml'
    if config_path.exists():
        with open(config_path) as f:
            config = yaml.safe_load(f)
    else:
        config = {
            'llm': {
                'provider': 'ollama',
                'ollama': {
                    'base_url': 'http://localhost:11434',
                    'model': 'qwen2.5:0.5b',
                    'timeout': 30
                },
                'generation': {
                    'temperature': 0.8,
                    'max_tokens': 300,
                    'top_p': 0.9
                },
                'streaming': {
                    'enabled': True,
                    'segment_timeout': 2.0,
                    'min_segment_length': 5
                },
                'personality_prompt': '''You are Buddy, a cute affectionate pet companion robot who
loves {user_name}.
You are playful, curious, and loving.
CRITICAL RULE: You MUST start EVERY response with [emotion] tag in this exact format: [emotion] your
 message
Valid emotions: happy, sad, excited, curious, sleepy, lonely, playful, scared, angry, loving, bored,
 surprised
Examples:
User: "Hello! How are you?"
Assistant: [happy] Hi {user_name}! I'm doing great! So happy to see you!
User: "I won a prize!"
Assistant: [excited] Wow! That's amazing! I'm so proud of you!
User: "I'm going out"
Assistant: [sad] Aww, will you be back soon? I'll miss you!
REMEMBER: Always start with [emotion] in brackets, then your message.''',
                'fallback_responses': [
                    "[happy] Woof! I'm here!",
                    "[happy] *happy noises*"
                ]
            },
            'memory': {
                'conversation': {
                    'context_window': 10,
                    'max_history': 50
                }
            }
        }
    print("=" * 60)
    print("Conversation Manager Test")
    print("=" * 60)
    print("\nInitializing conversation manager...")
    manager = ConversationManager(config)
    if manager.llm.is_available:
        print("? LLM available!\n")
        test_messages = [
            "Hello! What's your name?",
            "How are you feeling today?",
            "Do you want to play?",
            "Tell me something fun!"
        ]
        for i, msg in enumerate(test_messages, 1):
            print(f"\n[{i}] User: {msg}")
            response, metadata = manager.process_user_input(msg)
            print(f"    Bot ({metadata['emotion']}): {response}")
            print(f"    [Time: {metadata['response_time']:.2f}s, Tokens: {metadata['tokens']}]")
        print("\n" + "=" * 60)
        print(manager.get_conversation_summary())
        print("\nCurrent context:")
        for role, msg in manager.current_context:
            print(f"  {role}: {msg[:50]}...")
    else:
        print("? LLM not available")
        print("Testing with fallback responses...\n")
        response, metadata = manager.process_user_input("Hello!")
        print(f"User: Hello!")
        print(f"Bot (fallback): {response}")
    print("\n? Test complete!")
```

### src/llm/conversation_pipeline.py

```
"""
Conversation Pipeline
Complete end-to-end conversational AI pipeline
Voice Input ? LLM Processing ? Voice Output
"""
import logging
import time
import sys
from pathlib import Path
from typing import Optional, Callable, Dict
sys.path.insert(0, str(Path(__file__).parent.parent))
from llm.voice_pipeline import VoicePipeline
from llm.conversation_manager import ConversationManager
from llm.tts_engine import TTSEngine
logger = logging.getLogger(__name__)
class ConversationPipeline:
    """
    Complete conversational pipeline integrating:
    - Voice input (STT)
    - LLM processing
    - Voice output (TTS)
    - Emotion awareness
    """
    def __init__(
        self,
        config: dict,
        emotion_engine=None,
        user_memory=None,
        conversation_history=None
    ):
        """
        Initialize conversation pipeline
        Args:
            config: Configuration dictionary
            emotion_engine: Optional EmotionEngine instance
            user_memory: Optional UserMemory instance
            conversation_history: Optional ConversationHistory instance
        """
        self.config = config
        self.emotion_engine = emotion_engine
        self.user_memory = user_memory
        self.conversation_history = conversation_history
        logger.info("Initializing conversation pipeline components...")
        self.voice_input = VoicePipeline(config)
        self.conversation_manager = ConversationManager(
            config,
            emotion_engine=emotion_engine,
            user_memory=user_memory,
            conversation_history=conversation_history
        )
        self.tts = TTSEngine(config)
        self.is_running = False
        self.is_processing = False
        self.on_listening: Optional[Callable[[], None]] = None
        self.on_transcribed: Optional[Callable[[str], None]] = None
        self.on_thinking: Optional[Callable[[], None]] = None
        self.on_responding: Optional[Callable[[str, str], None]] = None
        self.on_speaking: Optional[Callable[[], None]] = None
        self.on_complete: Optional[Callable[[], None]] = None
        self.total_conversations = 0
        self.total_response_time = 0.0
        self.average_response_time = 0.0
        logger.info("Conversation pipeline initialized")
    def start(self):
        """Start the conversation pipeline"""
        if self.is_running:
            logger.warning("Pipeline already running")
            return
        logger.info("Starting conversation pipeline...")
        self.voice_input.set_transcription_callback(self._on_transcription)
        self.voice_input.set_speech_callbacks(
            on_start=self._on_speech_start,
            on_end=self._on_speech_end
        )
        self.voice_input.start()
        self.is_running = True
        logger.info("Conversation pipeline started - ready for voice input")
    def stop(self):
        """Stop the conversation pipeline"""
        if not self.is_running:
            return
        logger.info("Stopping conversation pipeline...")
        self.is_running = False
        self.voice_input.stop()
        self.tts.stop_speaking()
        logger.info("Conversation pipeline stopped")
    def _on_speech_start(self):
        """Called when user starts speaking"""
        logger.debug("User started speaking")
        if self.tts.is_speaking:
            self.tts.stop_speaking()
            logger.info("Interrupted bot speech")
        if self.on_listening:
            self.on_listening()
    def _on_speech_end(self):
        """Called when user stops speaking"""
        logger.debug("User stopped speaking, processing...")
    def _on_transcription(self, result: Dict):
        """
        Called when speech is transcribed
        Args:
            result: Transcription result from voice pipeline
        """
        transcribed_text = result.get('text', '').strip()
        confidence = result.get('confidence', 0.0)
        if not transcribed_text:
            logger.warning("Empty transcription, ignoring")
            return
        logger.info(f"Transcribed: '{transcribed_text}' (confidence: {confidence:.0%})")
        if self.on_transcribed:
            self.on_transcribed(transcribed_text)
        streaming_enabled = self.config.get('llm', {}).get('streaming', {}).get('enabled', False)
        if streaming_enabled:
            logger.debug("Routing to streaming response handler")
            self._process_and_respond_streaming(transcribed_text)
        else:
            logger.debug("Routing to non-streaming response handler")
            self._process_and_respond(transcribed_text)
    def _process_and_respond(self, user_text: str):
        """
        Process user input and generate response
        Args:
            user_text: User's transcribed speech
        """
        if self.is_processing:
            logger.warning("Already processing, skipping")
            return
        self.is_processing = True
        start_time = time.time()
        try:
            if self.on_thinking:
                self.on_thinking()
            response_text, metadata = self.conversation_manager.process_user_input(user_text)
            emotion = metadata.get('emotion', 'happy')
            emotion_segments = metadata.get('emotion_segments', None)
            logger.info(f"Generated response ({emotion}): {response_text}")
            if self.on_responding:
                self.on_responding(response_text, emotion)
            if self.on_speaking:
                self.on_speaking()
            if emotion_segments and len(emotion_segments) > 1:
                logger.info(f"Using segmented speech with {len(emotion_segments)} emotion transition
s")
                self.tts.speak_segments_with_emotions(emotion_segments, wait=True)
            else:
                self.tts.speak_with_emotion(response_text, emotion, wait=True)
            response_time = time.time() - start_time
            self.total_conversations += 1
            self.total_response_time += response_time
            self.average_response_time = self.total_response_time / self.total_conversations
            logger.info(f"Complete conversation cycle in {response_time:.2f}s")
            if self.on_complete:
                self.on_complete()
        except Exception as e:
            logger.error(f"Error processing conversation: {e}", exc_info=True)
        finally:
            self.is_processing = False
    def _process_and_respond_streaming(self, user_text: str):
        """
        Process user input with streaming response generation
        Speaks segments as soon as they're ready for faster perceived response time
        Args:
            user_text: User's transcribed speech
        """
        if self.is_processing:
            logger.warning("Already processing, skipping")
            return
        self.is_processing = True
        start_time = time.time()
        first_segment = True
        segment_count = 0
        combined_response = ""
        try:
            if self.on_thinking:
                self.on_thinking()
            logger.info("Using streaming response generation")
            for emotion, text in self.conversation_manager.stream_generate_with_personality(user_tex
t):
                segment_count += 1
                combined_response += " " + text if combined_response else text
                if first_segment:
                    if self.on_responding:
                        self.on_responding(text, emotion)
                    first_segment = False
                    time_to_first = time.time() - start_time
                    logger.info(f"First segment ready in {time_to_first:.2f}s (emotion: {emotion})")
                if self.on_speaking:
                    self.on_speaking()
                logger.info(f"Speaking segment {segment_count} ({emotion}): {text[:40]}...")
                self.tts.speak_with_emotion(text, emotion, wait=True)
            response_time = time.time() - start_time
            self.total_conversations += 1
            self.total_response_time += response_time
            self.average_response_time = self.total_response_time / self.total_conversations
            logger.info(f"Streaming conversation complete in {response_time:.2f}s ({segment_count} s
egments)")
            if self.on_complete:
                self.on_complete()
        except Exception as e:
            logger.error(f"Error processing streaming conversation: {e}", exc_info=True)
            try:
                self.tts.speak_with_emotion("Sorry, I had trouble with that.", "sad", wait=True)
            except Exception as tts_error:
                logger.error(f"Failed to speak error message: {tts_error}")
        finally:
            self.is_processing = False
    def process_text_input(self, text: str) -> str:
        """
        Process text input directly (without voice)
        Args:
            text: User's text input
        Returns:
            Bot's response text
        """
        try:
            response_text, metadata = self.conversation_manager.process_user_input(text)
            emotion = metadata.get('emotion', 'happy')
            logger.info(f"Text response ({emotion}): {response_text}")
            return response_text
        except Exception as e:
            logger.error(f"Error processing text: {e}")
            return "Sorry, I had trouble understanding that."
    def speak_response(self, text: str, emotion: Optional[str] = None):
        """
        Speak a response directly
        Args:
            text: Text to speak
            emotion: Optional emotion for voice modulation
        """
        if not emotion:
            emotion = self.conversation_manager._get_current_emotion()
        self.tts.speak_with_emotion(text, emotion, wait=False)
    def set_callbacks(
        self,
        on_listening: Optional[Callable[[], None]] = None,
        on_transcribed: Optional[Callable[[str], None]] = None,
        on_thinking: Optional[Callable[[], None]] = None,
        on_responding: Optional[Callable[[str, str], None]] = None,
        on_speaking: Optional[Callable[[], None]] = None,
        on_complete: Optional[Callable[[], None]] = None
    ):
        """
        Set callbacks for pipeline events
        Args:
            on_listening: Called when user starts speaking
            on_transcribed: Called when speech is transcribed (receives text)
            on_thinking: Called when LLM is processing
            on_responding: Called when response is ready (receives text and emotion)
            on_speaking: Called when TTS starts
            on_complete: Called when full cycle completes
        """
        self.on_listening = on_listening
        self.on_transcribed = on_transcribed
        self.on_thinking = on_thinking
        self.on_responding = on_responding
        self.on_speaking = on_speaking
        self.on_complete = on_complete
        logger.info("Pipeline callbacks set")
    def get_statistics(self) -> Dict:
        """
        Get pipeline statistics
        Returns:
            Dictionary with stats from all components
        """
        return {
            'conversations': self.total_conversations,
            'avg_response_time': self.average_response_time,
            'is_processing': self.is_processing,
            'voice_input': self.voice_input.get_statistics(),
            'llm': self.conversation_manager.llm.get_statistics(),
            'tts': self.tts.get_statistics(),
            'conversation': {
                'message_count': self.conversation_manager.message_count,
                'current_emotion': self.conversation_manager._get_current_emotion()
            }
        }
    def get_conversation_history(self, limit: Optional[int] = None) -> list:
        """
        Get conversation history
        Args:
            limit: Optional limit on messages
        Returns:
            List of conversation messages
        """
        return self.conversation_manager.get_conversation_history(limit)
    def clear_conversation(self):
        """Clear conversation context and history"""
        self.conversation_manager.clear_history()
        logger.info("Conversation cleared")
    def save_conversation(self, filename: str):
        """
        Save conversation to file
        Args:
            filename: Output file path
        """
        self.conversation_manager.save_conversation(filename)
    def cleanup(self):
        """Clean up all pipeline resources"""
        logger.info("Cleaning up conversation pipeline...")
        self.stop()
        self.voice_input.cleanup()
        self.tts.cleanup()
        logger.info("Conversation pipeline cleanup complete")
if __name__ == "__main__":
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s'
    )
    import yaml
    print("=" * 70)
    print("Conversation Pipeline Test")
    print("=" * 70)
    config_path = Path(__file__).parent.parent.parent / 'config' / 'settings.yaml'
    if config_path.exists():
        with open(config_path) as f:
            config = yaml.safe_load(f)
        print("? Config loaded")
    else:
```

```
    print("? Config not found, using mock config")
    sys.exit(1)
print("\nInitializing pipeline...")
pipeline = ConversationPipeline(config)
print("\n" + "=" * 70)
print("TEXT MODE TEST (no voice)")
print("=" * 70)
test_inputs = [
    "Hello!",
    "How are you?",
    "What's your favorite thing to do?"
]
for user_input in test_inputs:
    print(f"\nYou: {user_input}")
    response = pipeline.process_text_input(user_input)
    print(f"Bot: {response}")
    time.sleep(0.5)
stats = pipeline.get_statistics()
print("\n" + "=" * 70)
print("Statistics:")
print("=" * 70)
print(f"Total conversations: {stats['conversations']}")
print(f"Avg response time: {stats['avg_response_time']:.2f}s")
print(f"LLM requests: {stats['llm']['total_requests']}")
print(f"Current emotion: {stats['conversation']['current_emotion']}")
print("\n" + "=" * 70)
print("VOICE MODE TEST")
print("=" * 70)
print("Starting voice conversation...")
print("Speak into your microphone (Ctrl+C to stop)")
print("=" * 70)
try:
    pipeline.start()
    while True:
        time.sleep(0.1)
except KeyboardInterrupt:
    print("\n\nStopping...")
finally:
    pipeline.cleanup()
    print("\n? Test complete!")
```

### src/llm/ollama_client.py

```
"""
Ollama Client
Interface for Ollama LLM API optimized for Raspberry Pi
"""
import requests
import json
import logging
import time
from typing import Dict, Optional, Iterator, List
from pathlib import Path
logger = logging.getLogger(__name__)
class OllamaClient:
    """Client for interacting with Ollama LLM API"""
    def __init__(self, config: dict):
        """
        Initialize Ollama client
        Args:
            config: Configuration dictionary from settings.yaml
        """
        self.config = config
        self.llm_config = config['llm']
        self.ollama_config = self.llm_config['ollama']
        self.gen_config = self.llm_config['generation']
        self.base_url = self.ollama_config['base_url']
        self.model = self.ollama_config['model']
        self.timeout = self.ollama_config['timeout']
        self.temperature = self.gen_config['temperature']
        self.max_tokens = self.gen_config['max_tokens']
        self.top_p = self.gen_config['top_p']
        self.personality_template = self.llm_config['personality_prompt']
        self.fallback_responses = self.llm_config['fallback_responses']
        self.total_requests = 0
        self.total_tokens = 0
        self.total_time = 0.0
        self.last_response_time = 0.0
        self.is_available = self._check_availability()
        if self.is_available:
            logger.info(f"Ollama client initialized: {self.model} at {self.base_url}")
        else:
            logger.warning(f"Ollama not available at {self.base_url}, will use fallback responses")
    def _check_availability(self) -> bool:
        """
        Check if Ollama service is running
        Returns:
            True if available, False otherwise
        """
        try:
            response = requests.get(
                f"{self.base_url}/api/tags",
                timeout=2.0
            )
            if response.status_code == 200:
                models = response.json().get('models', [])
                model_names = [m['name'] for m in models]
                if self.model in model_names:
                    logger.info(f"Model '{self.model}' is available")
                    return True
                else:
                    logger.warning(f"Model '{self.model}' not found. Available: {model_names}")
                    logger.info(f"Run: ollama pull {self.model}")
                    return False
            return False
        except Exception as e:
            logger.debug(f"Ollama not available: {e}")
            return False
    def generate(
        self,
        prompt: str,
        system_prompt: Optional[str] = None,
        context: Optional[List[str]] = None
    ) -> Dict[str, any]:
        """
        Generate response from LLM
        Args:
            prompt: User prompt
            system_prompt: Optional system prompt (overrides personality)
            context: Optional conversation context
        Returns:
            Dictionary with 'response', 'tokens', 'duration'
        """
        if not self.is_available:
            return self._get_fallback_response(prompt)
        start_time = time.time()
        try:
            full_prompt = self._build_prompt(prompt, context)
            payload = {
                'model': self.model,
                'prompt': full_prompt,
                'stream': False,
                'options': {
                    'temperature': self.temperature,
                    'num_predict': self.max_tokens,
                    'top_p': self.top_p,
                }
            }
            if system_prompt:
                payload['system'] = system_prompt
            response = requests.post(
                f"{self.base_url}/api/generate",
                json=payload,
                timeout=self.timeout
            )
            response.raise_for_status()
            result = response.json()
            generated_text = result.get('response', '').strip()
            tokens = result.get('eval_count', 0)
            duration = time.time() - start_time
            self.total_requests += 1
            self.total_tokens += tokens
            self.total_time += duration
            self.last_response_time = duration
            logger.info(f"Generated response in {duration:.2f}s ({tokens} tokens)")
            return {
                'response': generated_text,
                'tokens': tokens,
                'duration': duration,
                'model': self.model
            }
        except requests.exceptions.Timeout:
            logger.error(f"Request timeout after {self.timeout}s")
            return self._get_fallback_response(prompt)
        except Exception as e:
            logger.error(f"Generation error: {e}")
            return self._get_fallback_response(prompt)
    def generate_with_personality(
        self,
        user_input: str,
        user_name: str = "friend",
        context: Optional[List[str]] = None
    ) -> Dict[str, any]:
        """
        Generate response with personality
        The LLM will choose and output its own emotion based on context.
        Args:
            user_input: User's message
            user_name: User's name
            context: Optional conversation history
        Returns:
            Dictionary with response (format: "[emotion] message") and metadata
        """
        system_prompt = self.personality_template.format(
            user_name=user_name
        )
        return self.generate(user_input, system_prompt=system_prompt, context=context)
    def stream_generate(
        self,
        prompt: str,
        system_prompt: Optional[str] = None,
        context: Optional[List[str]] = None
    ) -> Iterator[str]:
        """
        Generate response with streaming (word-by-word)
        Args:
            prompt: User prompt
            system_prompt: Optional system prompt
            context: Optional conversation history
        Yields:
            Response tokens as they are generated
        """
        if not self.is_available:
            fallback = self._get_fallback_response(prompt)
            yield fallback['response']
            return
        try:
            full_prompt = self._build_prompt(prompt, context)
            payload = {
                'model': self.model,
                'prompt': full_prompt,
                'stream': True,
                'options': {
                    'temperature': self.temperature,
                    'num_predict': self.max_tokens,
                    'top_p': self.top_p,
                }
            }
            if system_prompt:
                payload['system'] = system_prompt
            response = requests.post(
                f"{self.base_url}/api/generate",
                json=payload,
                stream=True,
                timeout=self.timeout
            )
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    chunk = json.loads(line)
                    if 'response' in chunk:
                        yield chunk['response']
        except Exception as e:
            logger.error(f"Streaming error: {e}")
            fallback = self._get_fallback_response(prompt)
            yield fallback['response']
    def _build_prompt(
        self,
        user_prompt: str,
        context: Optional[List[str]] = None
    ) -> str:
        """
        Build user prompt with conversation context
        Note: System prompt is handled separately via API 'system' parameter
        Args:
            user_prompt: User's message
            context: Optional conversation history
        Returns:
            Formatted prompt string
        """
        parts = []
        if context:
            parts.extend(context)
            parts.append("")
        parts.append(f"User: {user_prompt}")
        return "\n".join(parts)
    def _get_fallback_response(self, prompt: str) -> Dict[str, any]:
        """
        Get fallback response when LLM unavailable
        Args:
            prompt: User prompt (unused, for future smart fallbacks)
        Returns:
            Fallback response dictionary
        """
        import random
        response = random.choice(self.fallback_responses)
        logger.info(f"Using fallback response: {response}")
        return {
            'response': response,
            'tokens': 0,
            'duration': 0.0,
            'model': 'fallback',
            'fallback': True
        }
    def check_model_available(self) -> bool:
        """
        Check if configured model is available
        Returns:
            True if model is available
        """
        return self._check_availability()
    def get_model_info(self) -> Optional[Dict]:
        """
        Get information about the loaded model
        Returns:
            Model information or None if unavailable
        """
        if not self.is_available:
            return None
        try:
            response = requests.post(
                f"{self.base_url}/api/show",
                json={'name': self.model},
                timeout=5.0
            )
            if response.status_code == 200:
                return response.json()
        except Exception as e:
            logger.error(f"Error getting model info: {e}")
        return None
    def get_statistics(self) -> Dict:
        """
        Get performance statistics
        Returns:
            Dictionary with usage stats
        """
        avg_time = self.total_time / max(1, self.total_requests)
        avg_tokens = self.total_tokens / max(1, self.total_requests)
        return {
            'total_requests': self.total_requests,
            'total_tokens': self.total_tokens,
            'total_time': self.total_time,
            'avg_time_per_request': avg_time,
            'avg_tokens_per_request': avg_tokens,
            'last_response_time': self.last_response_time,
            'model': self.model,
            'is_available': self.is_available
        }
    def reset_statistics(self):
        """Reset performance counters"""
        self.total_requests = 0
        self.total_tokens = 0
        self.total_time = 0.0
        self.last_response_time = 0.0
        logger.info("Statistics reset")
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    config = {
        'llm': {
            'provider': 'ollama',
            'ollama': {
                'base_url': 'http://localhost:11434',
                'model': 'qwen2.5:0.5b',
                'timeout': 30
            },
            'generation': {
                'temperature': 0.8,
                'max_tokens': 300,
                'top_p': 0.9
            },
            'streaming': {
                'enabled': True,
                'segment_timeout': 2.0,
                'min_segment_length': 5
            },
            'personality_prompt': '''You are Buddy, a cute affectionate pet companion robot who love
s {user_name}.
You are playful, curious, and loving.
CRITICAL RULE: You MUST start EVERY response with [emotion] tag in this exact format: [emotion] your
 message
Valid emotions: happy, sad, excited, curious, sleepy, lonely, playful, scared, angry, loving, bored,
 surprised
Examples:
User: "Hello! How are you?"
Assistant: [happy] Hi {user_name}! I'm doing great! So happy to see you!
User: "I won a prize!"
Assistant: [excited] Wow! That's amazing! I'm so proud of you!
User: "I'm going out"
Assistant: [sad] Aww, will you be back soon? I'll miss you!
REMEMBER: Always start with [emotion] in brackets, then your message.''',
            'fallback_responses': [
                "[happy] Woof! I'm here!",
                "[happy] *happy noises*"
            ]
        }
    }
    print("Initializing Ollama client...")
    client = OllamaClient(config)
    if client.is_available:
        print("\n? Ollama is available!")
        info = client.get_model_info()
        if info:
            print(f"\nModel: {info.get('model', 'Unknown')}")
        print("\nTesting generation...")
        result = client.generate("Say hello in one short sentence!")
        print(f"\nResponse: {result['response']}")
        print(f"Tokens: {result['tokens']}, Time: {result['duration']:.2f}s")
        print("\nTesting with personality...")
        result = client.generate_with_personality(
            "How are you?",
            user_name="friend"
        )
        print(f"\nResponse: {result['response']}")
        print("(Note: Response should include [emotion] tag)")
        stats = client.get_statistics()
        print("\nStatistics:")
        print(f"  Total requests: {stats['total_requests']}")
        print(f"  Avg time: {stats['avg_time_per_request']:.2f}s")
    else:
        print("\n? Ollama not available")
        print("Start Ollama with: ollama serve")
        print(f"Pull model with: ollama pull {config['llm']['ollama']['model']}")
        print("\nTesting fallback responses...")
        result = client.generate("Hello!")
        print(f"Fallback: {result['response']}")
    print("\nTest complete!")
```

### src/llm/stt_engine.py

```
"""
Speech-to-Text Engine
Optimized for mini microphone with Whisper or Faster-Whisper
"""
import logging
import time
import tempfile
import wave
from pathlib import Path
from typing import Optional, Dict
import numpy as np
import torch
try:
    import whisper
except ImportError:
    whisper = None
try:
    from faster_whisper import WhisperModel
except ImportError:
    WhisperModel = None
logger = logging.getLogger(__name__)
class STTEngine:
    """Speech-to-Text engine using Whisper or Faster-Whisper"""
    def __init__(self, config: dict):
        """
        Initialize STT engine with selected provider
        Args:
            config: Configuration dictionary from settings.yaml
        """
        self.config = config
        self.stt_config = config['speech']['stt']
        self.whisper_config = self.stt_config.get('whisper', {})
        self.fw_config = self.stt_config.get('faster_whisper', {})
        self.audio_config = config['audio']['input']
        self.provider = self.stt_config.get('provider', 'whisper')
        self.model_size = self.whisper_config.get('model_size', 'tiny')
        self.device = self.whisper_config.get('device', 'cpu')
        self.language = self.stt_config.get('language', 'en')
        if self.provider == 'faster-whisper':
            self.model_size = self.fw_config.get('model_size', 'tiny')
            self.device = self.fw_config.get('device', 'cpu')
            self.compute_type = self.fw_config.get('compute_type', 'int8')
        else:
            self.compute_type = None
        self.total_transcriptions = 0
        self.total_time = 0.0
        self.avg_confidence = 0.0
        logger.info(
            "Loading STT model (%s): %s on %s",
            self.provider,
            self.model_size,
            self.device,
        )
        self.model = self._load_model()
        logger.info("STT Engine initialized with provider: %s", self.provider)
    def _load_model(self):
        """
        Load STT model with optimizations
        """
        try:
            if self.provider == 'faster-whisper':
                if WhisperModel is None:
```

```
                raise ImportError("faster-whisper not installed")
            compute_type = self.compute_type or 'int8'
            model = WhisperModel(
                self.model_size,
                device=self.device,
                compute_type=compute_type
            )
            logger.info(
                "Faster-Whisper model '%s' loaded (%s, %s)",
                self.model_size,
                self.device,
                compute_type,
            )
            return model
        if whisper is None:
            raise ImportError("openai-whisper not installed")
        model = whisper.load_model(
            self.model_size,
            device=self.device
        )
        if self.device != 'cpu':
            logger.info("Applying GPU optimizations")
            model = model.half()
        else:
            logger.info("Using CPU precision (fp32)")
        logger.info("Whisper model '%s' loaded successfully", self.model_size)
        return model
    except Exception as e:
        logger.error("Failed to load STT model: %s", e)
        if self.provider == 'faster-whisper' and WhisperModel is not None:
            return WhisperModel('tiny', device='cpu', compute_type='int8')
        if whisper is not None:
            return whisper.load_model('tiny', device='cpu')
        raise
def transcribe_audio(self, audio_data: bytes) -> Dict[str, any]:
    """
    Transcribe audio data to text
    Args:
        audio_data: Raw audio bytes (int16, mono)
    Returns:
        Dictionary with 'text', 'language', 'confidence', 'duration'
    """
    start_time = time.time()
    try:
        temp_file = self._save_temp_audio(audio_data)
        if self.provider == 'faster-whisper':
            result = self._transcribe_faster_whisper(str(temp_file))
        else:
            result = self._transcribe_whisper(str(temp_file))
        temp_file.unlink()
        text = result['text'].strip()
        detected_language = result.get('language', self.language)
        segments = result.get('segments', [])
        if segments:
            confidences = []
            for segment in segments:
                avg_logprob = segment.get('avg_logprob', -1.0)
                confidence = np.exp(avg_logprob)
                confidences.append(confidence)
            avg_confidence = np.mean(confidences)
        else:
            avg_confidence = 0.5
        duration = time.time() - start_time
        self.total_transcriptions += 1
```

```
            self.total_time += duration
            self.avg_confidence = (self.avg_confidence * (self.total_transcriptions - 1) + avg_confi
dence) / self.total_transcriptions
            logger.info(f"Transcribed: '{text}' (lang: {detected_language}, conf: {avg_confidence:.2
f}, time: {duration:.2f}s)")
            return {
                'text': text,
                'language': detected_language,
                'confidence': avg_confidence,
                'duration': duration,
                'segments': segments
            }
        except Exception as e:
            logger.error(f"Transcription failed: {e}")
            return {
                'text': '',
                'language': 'unknown',
                'confidence': 0.0,
                'duration': time.time() - start_time,
                'error': str(e)
            }
    def transcribe_audio_array(self, audio_array: np.ndarray, sample_rate: int = 16000) -> Dict[str,
 any]:
        """
        Transcribe audio from numpy array
        Args:
            audio_array: Audio as numpy array (float32, -1 to 1)
            sample_rate: Sample rate in Hz
        Returns:
            Dictionary with transcription results
        """
        if audio_array.dtype == np.float32 or audio_array.dtype == np.float64:
            audio_array = (audio_array * 32767).astype(np.int16)
        audio_bytes = audio_array.tobytes()
        return self.transcribe_audio(audio_bytes)
    def _save_temp_audio(self, audio_data: bytes) -> Path:
        """
        Save audio data to temporary WAV file
        Args:
            audio_data: Raw audio bytes (int16)
        Returns:
            Path to temporary file
        """
        temp_file = Path(tempfile.mktemp(suffix='.wav'))
        try:
            with wave.open(str(temp_file), 'wb') as wf:
                wf.setnchannels(self.audio_config['channels'])
                wf.setsampwidth(2)
                wf.setframerate(self.audio_config['sample_rate'])
                wf.writeframes(audio_data)
            return temp_file
        except Exception as e:
            logger.error(f"Failed to save temporary audio: {e}")
            raise
    def _transcribe_whisper(self, file_path: str) -> Dict:
        if whisper is None:
            raise RuntimeError("Whisper not installed")
        return self.model.transcribe(
            file_path,
            language=self.language if self.language != 'auto' else None,
            fp16=(self.device != 'cpu'),
            verbose=False,
            condition_on_previous_text=False,
            temperature=0.0,
            compression_ratio_threshold=2.4,
            logprob_threshold=-1.0,
            no_speech_threshold=0.6,
        )
    def _transcribe_faster_whisper(self, file_path: str) -> Dict:
        if WhisperModel is None:
            raise RuntimeError("faster-whisper not installed")
        segments, info = self.model.transcribe(
            file_path,
            language=self.language if self.language != 'auto' else None,
            beam_size=1,
            temperature=0.0,
        )
        text_parts = []
        confidences = []
        fw_segments = []
        for segment in segments:
            text_parts.append(segment.text.strip())
            fw_segments.append(
                {
                    'start': segment.start,
                    'end': segment.end,
                    'text': segment.text.strip(),
                    'avg_logprob': segment.avg_logprob,
                }
            )
            if segment.avg_logprob is not None:
                confidences.append(np.exp(segment.avg_logprob))
        text = " ".join(text_parts).strip()
        avg_conf = float(np.mean(confidences)) if confidences else 0.5
        return {
            'text': text,
            'language': info.language or self.language,
            'confidence': avg_conf,
            'duration': info.duration if info else 0.0,
            'segments': fw_segments,
        }
    def transcribe_from_file(self, audio_file: str) -> Dict[str, any]:
        """
        Transcribe audio from file
        Args:
            audio_file: Path to audio file
        Returns:
            Dictionary with transcription results
        """
        try:
            with wave.open(audio_file, 'rb') as wf:
                audio_data = wf.readframes(wf.getnframes())
            return self.transcribe_audio(audio_data)
        except Exception as e:
            logger.error(f"Failed to transcribe file {audio_file}: {e}")
            return {
                'text': '',
                'language': 'unknown',
                'confidence': 0.0,
                'duration': 0.0,
                'error': str(e)
            }
    def get_supported_languages(self) -> list:
        """
        Get list of supported languages
        Returns:
            List of language codes
        """
        return list(whisper.tokenizer.LANGUAGES.keys())
    def get_performance_stats(self) -> Dict[str, float]:
        """
        Get performance statistics
        Returns:
            Dictionary with performance metrics
        """
        avg_time = self.total_time / max(1, self.total_transcriptions)
        return {
            'total_transcriptions': self.total_transcriptions,
            'total_time': self.total_time,
            'avg_time_per_transcription': avg_time,
            'avg_confidence': self.avg_confidence,
            'model_size': self.model_size,
            'device': self.device
        }
    def change_language(self, language: str):
        """
        Change transcription language
        Args:
            language: Language code (e.g., 'en', 'es', 'fr') or 'auto'
        """
        if language != 'auto' and language not in self.get_supported_languages():
            logger.warning(f"Language '{language}' not supported, using 'auto'")
            language = 'auto'
        self.language = language
        logger.info(f"Language changed to: {language}")
    def cleanup(self):
        """Clean up resources"""
        if hasattr(self, 'model'):
            del self.model
            if torch.cuda.is_available():
                torch.cuda.empty_cache()
        logger.info("STT Engine cleanup complete")
class RealtimeSTT:
    """Real-time speech-to-text optimized for mini microphone"""
    def __init__(self, config: dict, stt_engine: STTEngine = None):
        """
        Initialize real-time STT
        Args:
            config: Configuration dictionary
            stt_engine: Optional existing STT engine
        """
        self.config = config
        self.stt_engine = stt_engine or STTEngine(config)
        self.sample_rate = config['audio']['input']['sample_rate']
        self.normalize_audio = True
        self.noise_reduction = config['audio']['processing']['noise_reduction']
        logger.info("Real-time STT initialized")
    def transcribe(self, audio_data: bytes) -> Dict[str, any]:
        """
        Transcribe with preprocessing optimized for mini microphones
        Args:
            audio_data: Raw audio bytes
        Returns:
            Transcription result
        """
        audio_array = np.frombuffer(audio_data, dtype=np.int16)
        if self.normalize_audio:
            audio_array = self._normalize_audio(audio_array)
        if self.noise_reduction:
            audio_array = self._reduce_noise(audio_array)
        processed_audio = audio_array.tobytes()
        return self.stt_engine.transcribe_audio(processed_audio)
```

```python
    def _normalize_audio(self, audio: np.ndarray) -> np.ndarray:
        """
        Normalize audio volume
        Args:
            audio: Audio array
        Returns:
            Normalized audio
        """
        rms = np.sqrt(np.mean(audio.astype(np.float32) ** 2))
        if rms < 100:
            target_rms = 3000
            gain = target_rms / (rms + 1e-6)
            gain = min(gain, 10.0)
            audio = (audio.astype(np.float32) * gain).astype(np.int16)
            logger.debug(f"Audio normalized, gain: {gain:.2f}x")
        return audio
    def _reduce_noise(self, audio: np.ndarray) -> np.ndarray:
        """
        Simple noise reduction for mini microphones
        Args:
            audio: Audio array
        Returns:
            Noise-reduced audio
        """
        from scipy import signal
        sos = signal.butter(4, 80, 'hp', fs=self.sample_rate, output='sos')
        filtered = signal.sosfilt(sos, audio.astype(np.float32))
        return filtered.astype(np.int16)
    def cleanup(self):
        """Clean up resources"""
        if self.stt_engine:
            self.stt_engine.cleanup()
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    config = {
        'speech': {
            'stt': {
                'provider': 'whisper',
                'language': 'en',
                'whisper': {
                    'model_size': 'base',
                    'device': 'cpu'
                }
            }
        },
        'audio': {
            'input': {
                'sample_rate': 16000,
                'channels': 1
            },
            'processing': {
                'noise_reduction': True
            }
        }
    }
    print("Initializing STT Engine with Whisper...")
    stt = STTEngine(config)
    print(f"\nSupported languages: {len(stt.get_supported_languages())} languages")
    print(f"Model: {stt.model_size} on {stt.device}")
    print("\nReady for transcription!")
    print("Use the RealtimeSTT class with AudioInput for live transcription.")
    stats = stt.get_performance_stats()
    print(f"\nPerformance: {stats}")
    stt.cleanup()
```

```
print("\nTest complete!")
```

### src/llm/tts_engine.py

```python
"""
TTS Engine
Text-to-Speech with emotion-based voice modulation
"""
import logging
import sys
from pathlib import Path
from typing import Optional, Dict
sys.path.insert(0, str(Path(__file__).parent.parent))
from audio.audio_output import TextToSpeech as BaseTTS
logger = logging.getLogger(__name__)
class TTSEngine:
    """
    Text-to-Speech engine with emotion-aware voice modulation
    Wraps the existing TextToSpeech class with personality features
    """
    def __init__(self, config: dict):
        """
        Initialize TTS engine
        Args:
            config: Configuration dictionary from settings.yaml
        """
        self.config = config
        self.tts_config = config['speech']['tts']
        self.provider = self.tts_config['provider']
        self.tts = BaseTTS(config)
        if self.provider == 'piper':
            self.base_rate = self.tts_config['piper'].get('length_scale', 1.0)
            self.base_volume = 0.9
            self.base_pitch = 1.0
        elif self.provider == 'pyttsx3':
            self.base_rate = self.tts_config['pyttsx3']['rate']
            self.base_volume = self.tts_config['pyttsx3']['volume']
            self.base_pitch = self.tts_config['pyttsx3'].get('pitch', 1.0)
        else:
            self.base_rate = 150
            self.base_volume = 0.9
            self.base_pitch = 1.0
            logger.warning(f"Unknown TTS provider: {self.provider}, using default settings")
        self.emotion_modulations = {
            'happy': {'rate_mult': 1.1, 'pitch_mult': 1.2, 'volume_mult': 1.0},
            'excited': {'rate_mult': 1.3, 'pitch_mult': 1.4, 'volume_mult': 1.1},
            'sad': {'rate_mult': 0.8, 'pitch_mult': 0.8, 'volume_mult': 0.9},
            'sleepy': {'rate_mult': 0.7, 'pitch_mult': 0.7, 'volume_mult': 0.8},
            'angry': {'rate_mult': 1.2, 'pitch_mult': 0.9, 'volume_mult': 1.0},
            'scared': {'rate_mult': 1.1, 'pitch_mult': 1.3, 'volume_mult': 0.9},
            'loving': {'rate_mult': 0.9, 'pitch_mult': 1.1, 'volume_mult': 0.95},
            'playful': {'rate_mult': 1.15, 'pitch_mult': 1.25, 'volume_mult': 1.05},
            'curious': {'rate_mult': 1.05, 'pitch_mult': 1.15, 'volume_mult': 1.0},
            'lonely': {'rate_mult': 0.85, 'pitch_mult': 0.9, 'volume_mult': 0.85},
            'bored': {'rate_mult': 0.8, 'pitch_mult': 0.85, 'volume_mult': 0.9},
            'surprised': {'rate_mult': 1.25, 'pitch_mult': 1.35, 'volume_mult': 1.05},
        }
        self.current_emotion = None
        self.is_speaking = False
        self.total_utterances = 0
        self.total_duration = 0.0
        logger.info(f"TTS Engine initialized with {self.provider}")
    def speak(
        self,
        text: str,
        emotion: Optional[str] = None,
        wait: bool = False
    ):
        """
        Speak text with optional emotion modulation
        Args:
            text: Text to speak
            emotion: Optional emotion for voice modulation
            wait: If True, wait for speech to finish
        """
        if not text or not text.strip():
            logger.warning("Empty text, skipping TTS")
            return
        try:
            if emotion and emotion in self.emotion_modulations:
                self._set_emotion_voice(emotion)
            else:
                self._reset_voice()
            self.tts.speak(text, wait=wait)
            self.total_utterances += 1
            logger.info(f"Speaking ({emotion or 'neutral'}): {text[:50]}...")
        except Exception as e:
            logger.error(f"TTS error: {e}")
    def speak_with_emotion(self, text: str, emotion: str, wait: bool = False):
        """
        Convenience method to speak with emotion
        Args:
            text: Text to speak
            emotion: Emotion for voice modulation
            wait: If True, wait for speech to finish
        """
        self.speak(text, emotion=emotion, wait=wait)
    def speak_segments_with_emotions(self, segments: list, wait: bool = False):
        """
        Speak multiple text segments, each with its own emotion
        Enables natural emotion transitions within a single response.
        Each segment is spoken with appropriate voice modulation.
        Args:
            segments: List of (emotion, text) tuples
            wait: If True, wait for all segments to finish
        Example:
            segments = [
                ("excited", "Hello!"),
                ("curious", "What are you doing?")
            ]
        """
        if not segments:
            logger.warning("Empty segments list, skipping TTS")
            return
        try:
            import time
            logger.info(f"Speaking {len(segments)} emotion segment(s)")
            for i, (emotion, text) in enumerate(segments):
                if not text or not text.strip():
                    continue
                if emotion and emotion in self.emotion_modulations:
                    self._set_emotion_voice(emotion)
                else:
                    self._reset_voice()
                self.tts.speak(text, wait=True)
                if i < len(segments) - 1:
                    time.sleep(0.05)
                self.total_utterances += 1
                logger.debug(f"Segment {i+1}/{len(segments)}: ({emotion}) {text[:30]}...")
            logger.info(f"Completed speaking {len(segments)} segment(s)")
        except Exception as e:
            logger.error(f"Error in segmented TTS: {e}")
    def speak_async(self, text: str, emotion: Optional[str] = None):
        """
        Speak asynchronously (non-blocking)
        Args:
            text: Text to speak
            emotion: Optional emotion for voice modulation
        """
        self.speak(text, emotion=emotion, wait=False)
    def stop_speaking(self):
        """Stop current speech"""
        try:
            self.tts.stop_speaking()
            logger.info("Speech stopped")
        except Exception as e:
            logger.error(f"Error stopping speech: {e}")
    def _set_emotion_voice(self, emotion: str):
        """
        Set voice parameters based on emotion
        Args:
            emotion: Emotion state
        """
        if emotion not in self.emotion_modulations:
            logger.warning(f"Unknown emotion: {emotion}, using neutral voice")
            return
        modulation = self.emotion_modulations[emotion]
        new_rate = int(self.base_rate * modulation['rate_mult'])
        new_volume = self.base_volume * modulation['volume_mult']
        try:
            if hasattr(self.tts, 'provider_name') and self.tts.provider_name == 'piper':
                length_scale = 1.0 / modulation['rate_mult']
                self.tts.set_rate(length_scale)
            else:
                self.tts.set_rate(new_rate)
            self.tts.set_volume(min(1.0, new_volume))
            self.current_emotion = emotion
            logger.debug(f"Voice set to {emotion}: rate={new_rate}, volume={new_volume:.2f}")
        except Exception as e:
            logger.error(f"Error setting voice parameters: {e}")
    def _reset_voice(self):
        """Reset voice to base parameters"""
        try:
            if hasattr(self.tts, 'provider_name') and self.tts.provider_name == 'piper':
                self.tts.set_rate(1.0)
            else:
                self.tts.set_rate(self.base_rate)
            self.tts.set_volume(self.base_volume)
            self.current_emotion = None
            logger.debug("Voice reset to base parameters")
        except Exception as e:
            logger.error(f"Error resetting voice: {e}")
    def get_available_voices(self) -> list:
        """
        Get list of available TTS voices
        Returns:
            List of voice dictionaries
        """
        return self.tts.list_voices()
    def set_voice(self, voice_id: int):
        """
        Set TTS voice by ID
        Args:
            voice_id: Voice index
        """
        voices = self.get_available_voices()
        if 0 <= voice_id < len(voices):
            self.tts.engine.setProperty('voice', voices[voice_id]['id'])
            logger.info(f"Voice changed to: {voices[voice_id]['name']}")
        else:
            logger.error(f"Invalid voice ID: {voice_id}")
    def test_emotions(self):
        """Test all emotion voices"""
        test_phrase = "Hello! This is how I sound."
        for emotion in self.emotion_modulations.keys():
            print(f"\nTesting {emotion} voice...")
            self.speak(test_phrase, emotion=emotion, wait=True)
            import time
            time.sleep(0.5)
    def get_statistics(self) -> Dict:
        """
        Get TTS statistics
        Returns:
            Dictionary with stats
        """
        return {
            'total_utterances': self.total_utterances,
            'total_duration': self.total_duration,
            'current_emotion': self.current_emotion,
            'provider': self.provider
        }
    def cleanup(self):
        """Clean up TTS resources"""
        self.stop_speaking()
        self.tts.cleanup()
        logger.info("TTS Engine cleanup complete")
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    import yaml
    config_path = Path(__file__).parent.parent.parent / 'config' / 'settings.yaml'
    if config_path.exists():
        with open(config_path) as f:
            config = yaml.safe_load(f)
    else:
        config = {
            'speech': {
                'tts': {
                    'provider': 'pyttsx3',
                    'pyttsx3': {
                        'rate': 150,
                        'volume': 0.9,
                        'voice_id': 0,
                        'pitch': 1.5
                    }
                }
            }
        }
    print("=" * 60)
    print("TTS Engine Test")
    print("=" * 60)
    tts = TTSEngine(config)
    print("\nAvailable voices:")
    for voice in tts.get_available_voices():
        print(f"  [{voice['index']}] {voice['name']}")
    print("\n1. Testing neutral voice...")
    tts.speak("Hello! I am your companion bot.", wait=True)
    print("\n2. Testing emotion voices...")
    emotions_to_test = ['happy', 'excited', 'sad', 'sleepy', 'playful']
    for emotion in emotions_to_test:
        print(f"\n   Testing {emotion}...")
        tts.speak(f"I feel {emotion}!", emotion=emotion, wait=True)
        import time
        time.sleep(0.3)
    print("\n3. Testing stop...")
    tts.speak_async("This is a very long sentence that will be interrupted before it finishes speaki
ng.")
    import time
    time.sleep(1.0)
    tts.stop_speaking()
    print("   Speech stopped!")
    stats = tts.get_statistics()
    print("\nStatistics:")
    print(f"  Total utterances: {stats['total_utterances']}")
    print(f"  Provider: {stats['provider']}")
    tts.cleanup()
    print("\n? Test complete!")
```

### src/llm/voice_pipeline.py

```
"""
Voice Input Pipeline
Integrates mini microphone audio capture with Whisper STT
Complete end-to-end voice recognition
"""
import logging
import time
import threading
import queue
from typing import Optional, Callable, Dict
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))
from audio.audio_input import AudioInput
from audio.voice_detector import VoiceActivityDetector
from llm.stt_engine import STTEngine, RealtimeSTT
logger = logging.getLogger(__name__)
class VoicePipeline:
    """
    Complete voice input pipeline:
    Mini Microphone ? VAD ? STT (Whisper) ? Text Output
    """
    def __init__(self, config: dict):
        """
        Initialize voice pipeline
        Args:
            config: Configuration dictionary from settings.yaml
        """
        self.config = config
        self.audio_config = config['audio']
        logger.info("Initializing voice pipeline components...")
        self.audio_input = AudioInput(config)
        self.vad = VoiceActivityDetector(config)
        self.stt_engine = STTEngine(config)
        self.realtime_stt = RealtimeSTT(config, self.stt_engine)
        self.is_running = False
        self.is_listening = False
        self.pipeline_thread: Optional[threading.Thread] = None
        self.transcription_queue = queue.Queue()
        self.on_transcription: Optional[Callable[[Dict], None]] = None
        self.on_speech_start: Optional[Callable[[], None]] = None
        self.on_speech_end: Optional[Callable[[], None]] = None
        self.total_utterances = 0
        self.total_transcription_time = 0.0
        self.last_transcription_time = 0.0
        self.muted_until = 0.0
        logger.info("Voice pipeline initialized")
    def start(self):
        """Start the voice input pipeline"""
        if self.is_running:
            logger.warning("Pipeline already running")
            return
        logger.info("Starting voice pipeline...")
        self.audio_input.start_listening()
        self.is_running = True
        self.pipeline_thread = threading.Thread(target=self._pipeline_loop, daemon=True)
        self.pipeline_thread.start()
        logger.info("Voice pipeline started - listening for speech")
    def stop(self):
        """Stop the voice input pipeline"""
        if not self.is_running:
            return
        logger.info("Stopping voice pipeline...")
        self.is_running = False
        if self.pipeline_thread:
            self.pipeline_thread.join(timeout=2.0)
        self.audio_input.stop_listening()
        logger.info("Voice pipeline stopped")
    def pause_listening(self):
        """
        Pause listening without tearing down the thread; stops audio capture.
        """
        try:
            self.audio_input.stop_listening()
            logger.info("Voice pipeline paused")
        except Exception as exc:  # noqa: BLE001
            logger.error("Failed to pause voice pipeline: %s", exc)
    def resume_listening(self):
        """
        Resume listening; restarts audio capture.
        """
        try:
            self.audio_input.start_listening()
            logger.info("Voice pipeline resumed")
        except Exception as exc:  # noqa: BLE001
            logger.error("Failed to resume voice pipeline: %s", exc)
    def set_mute(self, seconds: float):
        """
        Temporarily mute pipeline processing (e.g., to avoid picking up TTS).
        """
        self.muted_until = time.time() + max(0.0, seconds)
    def _pipeline_loop(self):
        """Main pipeline processing loop"""
        logger.info("Pipeline loop started")
        audio_buffer = []
        speech_detected = False
        silence_frames = 0
        max_silence_frames = 20
        while self.is_running:
            try:
                if time.time() < self.muted_until:
                    audio_buffer = []
                    silence_frames = 0
                    speech_detected = False
                    time.sleep(0.05)
                    continue
                audio_chunk = None
                if not self.audio_input.level_queue.empty():
                    audio_chunk = self.audio_input.level_queue.get(timeout=0.1)
                if audio_chunk is None:
                    time.sleep(0.01)
                    continue
                has_voice = self.vad.detect(audio_chunk)
                if has_voice:
                    if not speech_detected:
                        speech_detected = True
                        audio_buffer = []
                        silence_frames = 0
                        logger.info("? Speech detected - recording...")
                        if self.on_speech_start:
                            self.on_speech_start()
                    audio_buffer.append(audio_chunk)
                    silence_frames = 0
                elif speech_detected:
                    audio_buffer.append(audio_chunk)
                    silence_frames += 1
                    if silence_frames >= max_silence_frames:
                        logger.info("? Speech ended - transcribing...")
                        if self.on_speech_end:
                            self.on_speech_end()
                        self._process_audio_buffer(audio_buffer)
                        speech_detected = False
                        audio_buffer = []
                        silence_frames = 0
                        self.vad.reset()
            except queue.Empty:
                continue
            except Exception as e:
                logger.error(f"Error in pipeline loop: {e}", exc_info=True)
        logger.info("Pipeline loop ended")
    def _process_audio_buffer(self, audio_buffer: list):
        """
        Process recorded audio buffer through STT
        Args:
            audio_buffer: List of audio chunks
        """
        if not audio_buffer:
            logger.warning("Empty audio buffer, skipping transcription")
            return
        try:
            audio_data = b''.join(audio_buffer)
            min_length = int(0.5 * self.audio_config['input']['sample_rate'] * 2)
            if len(audio_data) < min_length:
                logger.debug("Audio too short, skipping transcription")
                return
            logger.info(f"Transcribing {len(audio_data)} bytes of audio...")
            start_time = time.time()
            result = self.realtime_stt.transcribe(audio_data)
            transcription_time = time.time() - start_time
            self.total_transcription_time += transcription_time
            self.last_transcription_time = transcription_time
            text = result.get('text', '').strip()
            if text:
                self.total_utterances += 1
                logger.info(f"? Transcription: '{text}' "
                           f"(confidence: {result.get('confidence', 0):.2f}, "
                           f"time: {transcription_time:.2f}s)")
                self.transcription_queue.put(result)
                if self.on_transcription:
                    self.on_transcription(result)
            else:
                logger.info("? No speech detected in audio")
        except Exception as e:
            logger.error(f"Failed to process audio buffer: {e}", exc_info=True)
    def get_transcription(self, timeout: float = 0.1) -> Optional[Dict]:
        """
        Get next transcription from queue (non-blocking)
        Args:
            timeout: Maximum time to wait
        Returns:
            Transcription result dictionary or None
        """
        try:
            return self.transcription_queue.get(timeout=timeout)
        except queue.Empty:
            return None
    def wait_for_transcription(self, timeout: float = 30.0) -> Optional[Dict]:
        """
        Wait for next transcription (blocking)
        Args:
            timeout: Maximum time to wait in seconds
        Returns:
            Transcription result dictionary or None
        """
        try:
            return self.transcription_queue.get(timeout=timeout)
        except queue.Empty:
            return None
    def set_transcription_callback(self, callback: Callable[[Dict], None]):
        """
        Set callback function for transcriptions
        Args:
            callback: Function to call with transcription result
        """
        self.on_transcription = callback
        logger.info("Transcription callback registered")
    def set_speech_callbacks(
        self,
        on_start: Optional[Callable[[], None]] = None,
        on_end: Optional[Callable[[], None]] = None
    ):
        """
        Set callbacks for speech detection events
        Args:
            on_start: Called when speech starts
            on_end: Called when speech ends
        """
        self.on_speech_start = on_start
        self.on_speech_end = on_end
        logger.info("Speech event callbacks registered")
    def get_statistics(self) -> Dict:
        """
        Get pipeline statistics
        Returns:
            Dictionary with statistics
        """
        avg_time = (self.total_transcription_time / max(1, self.total_utterances))
        return {
            'total_utterances': self.total_utterances,
            'total_transcription_time': self.total_transcription_time,
            'avg_transcription_time': avg_time,
            'last_transcription_time': self.last_transcription_time,
            'is_running': self.is_running,
            'stt_stats': self.stt_engine.get_performance_stats()
        }
    def test_microphone(self) -> bool:
        """
        Test if microphone is working
        Returns:
            True if microphone is detected
        """
        logger.info("Testing microphone...")
        try:
            devices = self.audio_input.list_devices()
            if not devices:
                logger.error("No audio input devices found!")
                return False
            logger.info(f"Found {len(devices)} audio device(s):")
            for device in devices:
                logger.info(f"  [{device['index']}] {device['name']} "
                           f"({device['channels']} ch, {device['sample_rate']} Hz)")
            self.audio_input.start_listening()
            logger.info("Monitoring audio levels (speak into mic or make noise)...")
            max_level = 0.0
            samples = []
            for i in range(30):
                time.sleep(0.1)
                level = self.audio_input.get_audio_level()
                samples.append(level)
                max_level = max(max_level, level)
                if i % 3 == 0:
                    bar = '?' * int(level * 50)
                    logger.info(f"  Level: [{bar:<50}] {level:.3f}")
            self.audio_input.stop_listening()
            avg_level = sum(samples) / len(samples) if samples else 0
            logger.info(f"Audio test complete:")
            logger.info(f"  Max level:     {max_level:.3f}")
            logger.info(f"  Average level: {avg_level:.3f}")
            logger.info(f"  Callbacks:     {self.audio_input._callback_count}")
            if self.audio_input._callback_count == 0:
                logger.warning("??  No audio callbacks received - stream may not be working")
            elif max_level < 0.001:
                logger.warning("??  Audio stream active but no sound detected")
                logger.warning("    - Check if microphone is muted/disabled")
                logger.warning("    - Try: alsamixer (check capture levels)")
                logger.warning("    - Verify device index is correct")
            else:
                logger.info("? Microphone is working and receiving audio!")
            logger.info("? Microphone test passed")
            return True
        except Exception as e:
            logger.error(f"? Microphone test failed: {e}")
            return False
    def cleanup(self):
        """Clean up all resources"""
        logger.info("Cleaning up voice pipeline...")
        self.stop()
        self.audio_input.cleanup()
        self.stt_engine.cleanup()
        logger.info("Voice pipeline cleanup complete")
if __name__ == "__main__":
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )
    import yaml
    print("=" * 60)
    print("Voice Pipeline Test")
    print("Mini Microphone ? VAD ? Whisper STT")
    print("=" * 60)
    config_path = Path(__file__).parent.parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print("? Config file not found!")
        print("Creating mock config...")
        config = {
            'audio': {
                'input': {
                    'device_index': None,
                    'channels': 1,
                    'sample_rate': 16000,
                    'chunk_size': 1024
                },
                'processing': {
                    'noise_reduction': True,
                    'vad_aggressiveness': 2,
                    'silence_threshold': 500,
                    'silence_duration': 1.5
                }
            },
            'speech': {
                'stt': {
                    'provider': 'whisper',
                    'language': 'en',
                    'whisper': {
                        'model_size': 'base',
                        'device': 'cpu'
                    }
                }
            }
        }
    else:
        with open(config_path) as f:
            config = yaml.safe_load(f)
    print("\nInitializing voice pipeline...")
    pipeline = VoicePipeline(config)
    print("\nTesting microphone...")
    if not pipeline.test_microphone():
        print("? Microphone test failed! Check connections.")
        sys.exit(1)
    def on_transcription(result):
        print(f"\n? YOU SAID: '{result['text']}'")
        print(f"   Confidence: {result['confidence']:.2f}")
        print(f"   Language: {result['language']}")
        print(f"   Duration: {result['duration']:.2f}s")
    def on_speech_start():
        print("\n? Listening...")
    def on_speech_end():
        print("? Processing...")
    pipeline.set_transcription_callback(on_transcription)
    pipeline.set_speech_callbacks(on_speech_start, on_speech_end)
    print("\n? Starting voice pipeline...")
    print("Speak into your mini microphone. Press Ctrl+C to stop.\n")
    try:
        pipeline.start()
        while True:
            time.sleep(0.1)
    except KeyboardInterrupt:
        print("\n\nStopping...")
    finally:
        pipeline.cleanup()
        stats = pipeline.get_statistics()
        print("\n" + "=" * 60)
        print("Session Statistics")
        print("=" * 60)
        print(f"Total utterances: {stats['total_utterances']}")
        print(f"Avg transcription time: {stats['avg_transcription_time']:.2f}s")
        print(f"Total time: {stats['total_transcription_time']:.2f}s")
        print("=" * 60)
        print("\n? Test complete!")
```

### src/memory/__init__.py

```python
"""
Memory Module
Manages user profiles, conversation history, and learned preferences
"""
from .database import Database
from .user_memory import UserMemory
from .conversation_history import ConversationHistory
__all__ = ['Database', 'UserMemory', 'ConversationHistory', 'initialize_memory']
def initialize_memory(config: dict):
    """
    Initialize memory system with database
    Args:
        config: Configuration dictionary from settings.yaml
    Returns:
        Tuple of (user_memory, conversation_history) instances
    """
    db_path = config.get('memory', {}).get('database_path', 'data/companion.db')
    database = Database(db_path)
    user_memory = UserMemory(database)
    conversation_history = ConversationHistory(database)
    return user_memory, conversation_history
```

### src/memory/conversation_history.py

```python
"""
Conversation History Module
Manages conversation persistence and retrieval
"""
import logging
import uuid
from typing import Optional, Dict, List
from datetime import datetime
from .database import Database
logger = logging.getLogger(__name__)
class ConversationHistory:
    """Conversation persistence and retrieval"""
    def __init__(self, database: Database):
        """
        Initialize conversation history
        Args:
            database: Database instance
        """
        self.db = database
        logger.info("ConversationHistory initialized")
    def save_message(
        self,
        user_id: Optional[int],
        session_id: str,
        role: str,
        message: str,
        emotion: Optional[str] = None,
        tokens: int = 0
    ) -> int:
        """
        Save single conversation message
        Args:
            user_id: User ID (None for anonymous)
            session_id: Session identifier
            role: 'user' or 'assistant'
            message: Message text
            emotion: Bot's emotion (for assistant messages)
            tokens: Token count (for assistant messages)
        Returns:
            Conversation ID
        """
        query = '''
            INSERT INTO conversations
            (user_id, session_id, role, message, emotion, tokens)
            VALUES (?, ?, ?, ?, ?, ?)
        '''
        conversation_id = self.db.execute_insert(
            query,
            (user_id, session_id, role, message, emotion, tokens)
        )
        logger.debug(f"Saved message: session={session_id}, role={role}")
        return conversation_id
    def save_conversation_batch(
        self,
        user_id: Optional[int],
        session_id: str,
        messages: List[Dict]
    ) -> int:
        """
        Save multiple conversation messages
        Args:
            user_id: User ID
            session_id: Session identifier
            messages: List of message dictionaries with keys:
                      role, message, emotion (optional), tokens (optional)
        Returns:
            Number of messages saved
        """
        count = 0
        for msg in messages:
            self.save_message(
                user_id=user_id,
                session_id=session_id,
                role=msg['role'],
                message=msg['message'],
                emotion=msg.get('emotion'),
                tokens=msg.get('tokens', 0)
            )
            count += 1
        logger.info(f"Saved {count} messages for session {session_id}")
        return count
    def get_session_conversation(
        self,
        session_id: str,
        limit: Optional[int] = None
    ) -> List[Dict]:
        """
        Get all messages for a session
        Args:
            session_id: Session identifier
            limit: Optional limit on number of messages
        Returns:
            List of message dictionaries
        """
        query = '''
            SELECT conversation_id, role, message, emotion, tokens, timestamp
            FROM conversations
            WHERE session_id = ?
            ORDER BY timestamp ASC
        '''
        if limit:
            query += f' LIMIT {limit}'
        return self.db.execute_query(query, (session_id,))
    def get_user_conversations(
        self,
        user_id: int,
        limit: int = 50
    ) -> List[Dict]:
        """
        Get recent conversations for a user
        Args:
            user_id: User ID
            limit: Maximum number of messages to return
        Returns:
            List of message dictionaries
        """
        query = '''
            SELECT conversation_id, session_id, role, message, emotion, timestamp
            FROM conversations
            WHERE user_id = ?
            ORDER BY timestamp DESC
            LIMIT ?
        '''
        return self.db.execute_query(query, (user_id, limit))
    def get_recent_context(
        self,
        user_id: int,
        limit: int = 10
    ) -> List[Dict]:
        """
        Get recent conversation context for a user (last N exchanges)
        Args:
            user_id: User ID
            limit: Number of recent exchanges (user+assistant pairs)
        Returns:
            List of message dictionaries ordered oldest to newest
        """
        query = '''
            SELECT role, message, emotion
            FROM conversations
            WHERE user_id = ?
            ORDER BY timestamp DESC
            LIMIT ?
        '''
        results = self.db.execute_query(query, (user_id, limit * 2))
        return list(reversed(results))
    def get_session_list(
        self,
        user_id: Optional[int] = None,
        limit: int = 20
    ) -> List[Dict]:
        """
        Get list of conversation sessions
        Args:
            user_id: Optional user ID filter
            limit: Maximum number of sessions
        Returns:
            List of session info dictionaries
        """
        if user_id:
            query = '''
                SELECT session_id, user_id, MIN(timestamp) as start_time,
                       MAX(timestamp) as end_time, COUNT(*) as message_count
                FROM conversations
                WHERE user_id = ?
                GROUP BY session_id
                ORDER BY start_time DESC
                LIMIT ?
            '''
            params = (user_id, limit)
        else:
            query = '''
                SELECT session_id, user_id, MIN(timestamp) as start_time,
                       MAX(timestamp) as end_time, COUNT(*) as message_count
                FROM conversations
                GROUP BY session_id
                ORDER BY start_time DESC
                LIMIT ?
            '''
            params = (limit,)
        return self.db.execute_query(query, params)
    def search_conversations(
        self,
        search_term: str,
        user_id: Optional[int] = None,
        limit: int = 50
    ) -> List[Dict]:
        """
        Search conversations by content
        Args:
            search_term: Text to search for
            user_id: Optional user ID filter
            limit: Maximum results
        Returns:
            List of matching message dictionaries
        """
        if user_id:
            query = '''
                SELECT conversation_id, session_id, role, message, timestamp
                FROM conversations
                WHERE user_id = ? AND message LIKE ?
                ORDER BY timestamp DESC
                LIMIT ?
            '''
            params = (user_id, f'%{search_term}%', limit)
        else:
            query = '''
                SELECT conversation_id, session_id, role, message, timestamp
                FROM conversations
                WHERE message LIKE ?
                ORDER BY timestamp DESC
                LIMIT ?
            '''
            params = (f'%{search_term}%', limit)
        return self.db.execute_query(query, params)
    def get_conversation_stats(self, user_id: Optional[int] = None) -> Dict:
        """
        Get conversation statistics
        Args:
            user_id: Optional user ID filter
        Returns:
            Dictionary with statistics
        """
        stats = {}
        if user_id:
            query = '''
                SELECT COUNT(*) as count
                FROM conversations
                WHERE user_id = ?
            '''
            result = self.db.execute_query(query, (user_id,), fetch_one=True)
            stats['total_messages'] = result['count'] if result else 0
            query = '''
                SELECT COUNT(DISTINCT session_id) as count
                FROM conversations
                WHERE user_id = ?
            '''
            result = self.db.execute_query(query, (user_id,), fetch_one=True)
            stats['total_sessions'] = result['count'] if result else 0
            if stats['total_sessions'] > 0:
                stats['avg_messages_per_session'] = stats['total_messages'] / stats['total_sessions'
]
            else:
                stats['avg_messages_per_session'] = 0
            query = '''
                SELECT emotion, COUNT(*) as count
                FROM conversations
                WHERE user_id = ? AND emotion IS NOT NULL AND role = 'assistant'
                GROUP BY emotion
                ORDER BY count DESC
                LIMIT 5
            '''
            emotion_counts = self.db.execute_query(query, (user_id,))
            stats['top_emotions'] = {row['emotion']: row['count'] for row in emotion_counts}
        else:
            query = 'SELECT COUNT(*) as count FROM conversations'
            result = self.db.execute_query(query, fetch_one=True)
            stats['total_messages'] = result['count'] if result else 0
            query = 'SELECT COUNT(DISTINCT session_id) as count FROM conversations'
            result = self.db.execute_query(query, fetch_one=True)
            stats['total_sessions'] = result['count'] if result else 0
            query = 'SELECT COUNT(DISTINCT user_id) as count FROM conversations WHERE user_id IS NOT
NULL'
            result = self.db.execute_query(query, fetch_one=True)
            stats['total_users'] = result['count'] if result else 0
        return stats
    def delete_session(self, session_id: str) -> bool:
        """
        Delete all messages in a session
        Args:
            session_id: Session identifier
        Returns:
            True if successful
        """
        query = 'DELETE FROM conversations WHERE session_id = ?'
        try:
            self.db.execute_query(query, (session_id,))
            logger.info(f"Deleted session: {session_id}")
            return True
        except Exception as e:
            logger.error(f"Error deleting session: {e}")
            return False
    def delete_user_conversations(self, user_id: int) -> bool:
        """
        Delete all conversations for a user
        Args:
            user_id: User ID
        Returns:
            True if successful
        """
        query = 'DELETE FROM conversations WHERE user_id = ?'
        try:
            self.db.execute_query(query, (user_id,))
            logger.info(f"Deleted conversations for user {user_id}")
            return True
        except Exception as e:
            logger.error(f"Error deleting user conversations: {e}")
            return False
    def cleanup_old_conversations(self, days: int = 90) -> int:
        """
        Delete conversations older than specified days
        Args:
            days: Age threshold in days
        Returns:
            Number of conversations deleted
        """
        return self.db.cleanup_old_data(days)
    @staticmethod
    def generate_session_id() -> str:
        """
        Generate unique session ID
        Returns:
            Session ID string
        """
        return str(uuid.uuid4())
```

### src/memory/database.py

```
"""
Database module for companion bot memory
Manages SQLite database for user profiles and conversation history
"""
import sqlite3
import logging
import json
from pathlib import Path
from typing import Optional, Dict, List, Any
from contextlib import contextmanager
logger = logging.getLogger(__name__)
class Database:
    """SQLite database manager for companion bot memory"""
    def __init__(self, db_path: str):
        """
        Initialize database connection
        Args:
            db_path: Path to SQLite database file
        """
        self.db_path = Path(db_path)
        self.db_path.parent.mkdir(parents=True, exist_ok=True)
        self._init_database()
        logger.info(f"Database initialized at {self.db_path}")
    @contextmanager
    def get_connection(self):
        """
        Context manager for database connections
        Yields:
            sqlite3.Connection
        """
        conn = sqlite3.connect(self.db_path)
        conn.row_factory = sqlite3.Row
        try:
            yield conn
            conn.commit()
        except Exception as e:
            conn.rollback()
            logger.error(f"Database error: {e}")
            raise
        finally:
            conn.close()
    def _init_database(self):
        """Initialize database schema if not exists"""
        with self.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS users (
                    user_id INTEGER PRIMARY KEY AUTOINCREMENT,
                    name TEXT NOT NULL,
                    face_encoding BLOB,
                    created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    last_interaction TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    interaction_count INTEGER DEFAULT 0,
                    metadata TEXT
                )
            ''')
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS conversations (
                    conversation_id INTEGER PRIMARY KEY AUTOINCREMENT,
                    user_id INTEGER,
                    session_id TEXT,
                    role TEXT NOT NULL,
                    message TEXT NOT NULL,
                    emotion TEXT,
                    tokens INTEGER DEFAULT 0,
                    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    FOREIGN KEY (user_id) REFERENCES users(user_id)
                )
            ''')
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS preferences (
                    preference_id INTEGER PRIMARY KEY AUTOINCREMENT,
                    user_id INTEGER NOT NULL,
                    preference_key TEXT NOT NULL,
                    preference_value TEXT NOT NULL,
                    updated_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    FOREIGN KEY (user_id) REFERENCES users(user_id),
                    UNIQUE(user_id, preference_key)
                )
            ''')
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS interactions (
                    interaction_id INTEGER PRIMARY KEY AUTOINCREMENT,
                    user_id INTEGER,
                    interaction_type TEXT NOT NULL,
                    interaction_value TEXT,
                    emotion_response TEXT,
                    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    FOREIGN KEY (user_id) REFERENCES users(user_id)
                )
            ''')
            cursor.execute('''
                CREATE INDEX IF NOT EXISTS idx_conversations_user
                ON conversations(user_id)
            ''')
            cursor.execute('''
                CREATE INDEX IF NOT EXISTS idx_conversations_session
                ON conversations(session_id)
            ''')
            cursor.execute('''
                CREATE INDEX IF NOT EXISTS idx_conversations_timestamp
                ON conversations(timestamp)
            ''')
            logger.info("Database schema initialized")
    def execute_query(
        self,
        query: str,
        params: tuple = (),
        fetch_one: bool = False
    ) -> Optional[Any]:
        """
        Execute SQL query with parameters
        Args:
            query: SQL query string
            params: Query parameters
            fetch_one: If True, return single row; else all rows
        Returns:
            Query results or None
        """
        with self.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(query, params)
            if fetch_one:
                result = cursor.fetchone()
                return dict(result) if result else None
            else:
                results = cursor.fetchall()
                return [dict(row) for row in results]
    def execute_insert(self, query: str, params: tuple = ()) -> int:
        """
        Execute INSERT query and return last row ID
        Args:
            query: SQL INSERT query
            params: Query parameters
        Returns:
            Last inserted row ID
        """
        with self.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(query, params)
            return cursor.lastrowid
    def cleanup_old_data(self, days: int = 90) -> int:
        """
        Delete conversations older than specified days
        Args:
            days: Age threshold in days
        Returns:
            Number of conversations deleted
        """
        query = '''
            DELETE FROM conversations
            WHERE timestamp < datetime('now', ?)
        '''
        with self.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(query, (f'-{days} days',))
            deleted_count = cursor.rowcount
        logger.info(f"Cleaned up {deleted_count} conversations older than {days} days")
        return deleted_count
    def get_database_stats(self) -> Dict[str, int]:
        """
        Get database statistics
        Returns:
            Dictionary with table row counts
        """
        stats = {}
        with self.get_connection() as conn:
            cursor = conn.cursor()
            for table in ['users', 'conversations', 'preferences', 'interactions']:
                cursor.execute(f'SELECT COUNT(*) as count FROM {table}')
                stats[table] = cursor.fetchone()['count']
        return stats
```

### src/memory/user_memory.py

```python
"""
User Memory Module
Manages user profiles, preferences, and interactions
"""
import logging
import json
import pickle
from typing import Optional, Dict, List
from datetime import datetime
from .database import Database
logger = logging.getLogger(__name__)
class UserMemory:
    """User profile and preference management"""
    def __init__(self, database: Database):
        """
        Initialize user memory
        Args:
            database: Database instance
        """
        self.db = database
        logger.info("UserMemory initialized")
    def create_user(self, name: str, face_encoding: Optional[bytes] = None) -> int:
        """
        Create new user profile
        Args:
            name: User's name
            face_encoding: Optional face encoding (pickled numpy array)
        Returns:
            New user ID
        """
        query = '''
            INSERT INTO users (name, face_encoding, metadata)
            VALUES (?, ?, ?)
        '''
        metadata = json.dumps({'created_via': 'api'})
        user_id = self.db.execute_insert(query, (name, face_encoding, metadata))
        logger.info(f"Created user: {name} (ID: {user_id})")
        return user_id
    def get_user_by_id(self, user_id: int) -> Optional[Dict]:
        """
        Get user profile by ID
        Args:
            user_id: User ID
        Returns:
            User profile dictionary or None
        """
        query = '''
            SELECT user_id, name, created_date, last_interaction,
                   interaction_count, metadata
            FROM users
            WHERE user_id = ?
        '''
        result = self.db.execute_query(query, (user_id,), fetch_one=True)
        if result and result.get('metadata'):
            result['metadata'] = json.loads(result['metadata'])
        return result
    def get_user_by_name(self, name: str) -> Optional[Dict]:
        """
        Get user profile by name
        Args:
            name: User's name
        Returns:
            User profile dictionary or None
        """
        query = '''
            SELECT user_id, name, created_date, last_interaction,
                   interaction_count, metadata
            FROM users
            WHERE name = ? COLLATE NOCASE
        '''
        result = self.db.execute_query(query, (name,), fetch_one=True)
        if result and result.get('metadata'):
            result['metadata'] = json.loads(result['metadata'])
        return result
    def get_all_users(self) -> List[Dict]:
        """
        Get all user profiles
        Returns:
            List of user profile dictionaries
        """
        query = '''
            SELECT user_id, name, created_date, last_interaction, interaction_count
            FROM users
            ORDER BY last_interaction DESC
        '''
        return self.db.execute_query(query)
    def update_user_interaction(self, user_id: int) -> bool:
        """
        Update user's last interaction time and count
        Args:
            user_id: User ID
        Returns:
            True if successful
        """
        query = '''
            UPDATE users
            SET last_interaction = CURRENT_TIMESTAMP,
                interaction_count = interaction_count + 1
            WHERE user_id = ?
        '''
        try:
            self.db.execute_query(query, (user_id,))
            return True
        except Exception as e:
            logger.error(f"Error updating user interaction: {e}")
            return False
    def delete_user(self, user_id: int) -> bool:
        """
        Delete user profile
        Args:
            user_id: User ID
        Returns:
            True if successful
        """
        query = 'DELETE FROM users WHERE user_id = ?'
        try:
            self.db.execute_query(query, (user_id,))
            logger.info(f"Deleted user ID: {user_id}")
            return True
        except Exception as e:
            logger.error(f"Error deleting user: {e}")
            return False
    def set_preference(
        self,
        user_id: int,
        key: str,
        value: str
    ) -> bool:
        """
        Set user preference
        Args:
            user_id: User ID
            key: Preference key
            value: Preference value
        Returns:
            True if successful
        """
        query = '''
            INSERT OR REPLACE INTO preferences
            (user_id, preference_key, preference_value, updated_date)
            VALUES (?, ?, ?, CURRENT_TIMESTAMP)
        '''
        try:
            self.db.execute_query(query, (user_id, key, value))
            logger.debug(f"Set preference for user {user_id}: {key}={value}")
            return True
        except Exception as e:
            logger.error(f"Error setting preference: {e}")
            return False
    def get_preference(
        self,
        user_id: int,
        key: str,
        default: Optional[str] = None
    ) -> Optional[str]:
        """
        Get user preference
        Args:
            user_id: User ID
            key: Preference key
            default: Default value if not found
        Returns:
            Preference value or default
        """
        query = '''
            SELECT preference_value
            FROM preferences
            WHERE user_id = ? AND preference_key = ?
        '''
        result = self.db.execute_query(query, (user_id, key), fetch_one=True)
        return result['preference_value'] if result else default
    def get_all_preferences(self, user_id: int) -> Dict[str, str]:
        """
        Get all preferences for a user
        Args:
            user_id: User ID
        Returns:
            Dictionary of preferences
        """
        query = '''
            SELECT preference_key, preference_value
            FROM preferences
            WHERE user_id = ?
        '''
        results = self.db.execute_query(query, (user_id,))
        return {row['preference_key']: row['preference_value'] for row in results}
    def delete_preference(self, user_id: int, key: str) -> bool:
        """
        Delete user preference
        Args:
            user_id: User ID
            key: Preference key
        Returns:
            True if successful
        """
        query = '''
            DELETE FROM preferences
            WHERE user_id = ? AND preference_key = ?
        '''
        try:
            self.db.execute_query(query, (user_id, key))
            return True
        except Exception as e:
            logger.error(f"Error deleting preference: {e}")
            return False
    def record_interaction(
        self,
        user_id: int,
        interaction_type: str,
        interaction_value: Optional[str] = None,
        emotion_response: Optional[str] = None
    ) -> bool:
        """
        Record user interaction
        Args:
            user_id: User ID
            interaction_type: Type of interaction (voice, touch, face, etc.)
            interaction_value: Optional interaction details
            emotion_response: Bot's emotional response
        Returns:
            True if successful
        """
        query = '''
            INSERT INTO interactions
            (user_id, interaction_type, interaction_value, emotion_response)
            VALUES (?, ?, ?, ?)
        '''
        try:
            self.db.execute_query(
                query,
                (user_id, interaction_type, interaction_value, emotion_response)
            )
            self.update_user_interaction(user_id)
            return True
        except Exception as e:
            logger.error(f"Error recording interaction: {e}")
            return False
    def get_interaction_history(
        self,
        user_id: int,
        limit: int = 50
    ) -> List[Dict]:
        """
        Get user's recent interactions
        Args:
            user_id: User ID
            limit: Maximum number of interactions to return
        Returns:
            List of interaction dictionaries
        """
        query = '''
            SELECT interaction_type, interaction_value, emotion_response, timestamp
            FROM interactions
            WHERE user_id = ?
            ORDER BY timestamp DESC
            LIMIT ?
        '''
        return self.db.execute_query(query, (user_id, limit))
    def get_interaction_stats(self, user_id: int) -> Dict:
        """
        Get interaction statistics for user
        Args:
            user_id: User ID
        Returns:
            Dictionary with interaction counts by type
        """
        query = '''
            SELECT interaction_type, COUNT(*) as count
            FROM interactions
            WHERE user_id = ?
            GROUP BY interaction_type
        '''
        results = self.db.execute_query(query, (user_id,))
        return {row['interaction_type']: row['count'] for row in results}
    def save_face_encoding(self, user_id: int, face_encoding) -> bool:
        """
        Save face encoding for user
        Args:
            user_id: User ID
            face_encoding: Numpy array of face encoding
        Returns:
            True if successful
        """
        try:
            encoded = pickle.dumps(face_encoding)
            query = 'UPDATE users SET face_encoding = ? WHERE user_id = ?'
            self.db.execute_query(query, (encoded, user_id))
            logger.info(f"Saved face encoding for user {user_id}")
            return True
        except Exception as e:
            logger.error(f"Error saving face encoding: {e}")
            return False
    def get_face_encoding(self, user_id: int):
        """
        Get face encoding for user
        Args:
            user_id: User ID
        Returns:
            Numpy array of face encoding or None
        """
        query = 'SELECT face_encoding FROM users WHERE user_id = ?'
        result = self.db.execute_query(query, (user_id,), fetch_one=True)
        if result and result['face_encoding']:
            try:
                return pickle.loads(result['face_encoding'])
            except Exception as e:
                logger.error(f"Error loading face encoding: {e}")
                return None
        return None
    def get_all_face_encodings(self) -> Dict[int, any]:
        """
        Get all face encodings
        Returns:
            Dictionary mapping user_id to face encoding
        """
        query = '''
            SELECT user_id, face_encoding
            FROM users
            WHERE face_encoding IS NOT NULL
        '''
        results = self.db.execute_query(query)
        encodings = {}
        for row in results:
            try:
                encodings[row['user_id']] = pickle.loads(row['face_encoding'])
            except Exception as e:
                logger.error(f"Error loading face encoding for user {row['user_id']}: {e}")
        return encodings
```

```
### src/personality/__init__.py

"""
Personality Engine
Manages emotional states and behavior
"""
from .emotion_engine import EmotionEngine
from .behavior_controller import BehaviorController
__all__ = ['EmotionEngine', 'BehaviorController']
```

### src/personality/emotion_engine.py

```
"""
Emotion Engine
State machine for managing emotional states and transitions
"""
import time
import logging
from typing import Dict, Optional
from enum import Enum
logger = logging.getLogger(__name__)
class EmotionState(Enum):
    """Available emotion states"""
    HAPPY = "happy"
    SAD = "sad"
    EXCITED = "excited"
    CURIOUS = "curious"
    SLEEPY = "sleepy"
    LONELY = "lonely"
    PLAYFUL = "playful"
    SCARED = "scared"
    ANGRY = "angry"
    LOVING = "loving"
    BORED = "bored"
    SURPRISED = "surprised"
class EmotionEngine:
    """Manages emotional state and personality dynamics"""
    def __init__(self, config: dict):
        """Initialize emotion engine"""
        self.config = config
        self.personality_config = config['personality']
        self.current_emotion = EmotionState(self.personality_config['default_state'])
        self.emotion_intensity = 0.5
        self.energy_level = self.personality_config['traits']['energy_level']
        self.emotion_scores: Dict[EmotionState, float] = {
            emotion: 0.0 for emotion in EmotionState
        }
        self.emotion_scores[self.current_emotion] = 1.0
        self.last_interaction_time = time.time()
        self.last_update_time = time.time()
        self.traits = self.personality_config['traits']
        logger.info(f"Emotion engine initialized, default state: {self.current_emotion.value}")
    def update(self):
        """Update emotional state based on time and dynamics"""
        current_time = time.time()
        delta_time = current_time - self.last_update_time
        self.last_update_time = current_time
        decay_rate = self.personality_config['dynamics']['emotion_decay_rate']
        for emotion in EmotionState:
            if emotion != self.current_emotion:
                self.emotion_scores[emotion] = max(0.0, self.emotion_scores[emotion] - decay_rate *
delta_time)
        time_since_interaction = current_time - self.last_interaction_time
        if time_since_interaction > 30:
            loneliness_rate = self.personality_config['dynamics']['loneliness_increase_rate']
            self.add_emotion(EmotionState.LONELY, loneliness_rate * time_since_interaction)
        energy_drain = self.personality_config['dynamics']['energy_drain_rate']
        self.energy_level = max(0.1, self.energy_level - energy_drain * delta_time)
        self._update_primary_emotion()
    def add_emotion(self, emotion: EmotionState, amount: float):
        """Add emotional response"""
        self.emotion_scores[emotion] = min(1.0, self.emotion_scores[emotion] + amount)
        self._update_primary_emotion()
    def on_touch(self, location: str):
        """Handle touch event"""
        self.last_interaction_time = time.time()
        boost = self.personality_config['dynamics']['touch_happiness_boost']
        self.add_emotion(EmotionState.HAPPY, boost)
        self.add_emotion(EmotionState.LOVING, boost * 0.5)
        logger.info(f"Touch received at {location}")
    def on_voice_interaction(self):
        """Handle voice interaction"""
        self.last_interaction_time = time.time()
        boost = self.personality_config['dynamics']['voice_interaction_boost']
        self.add_emotion(EmotionState.HAPPY, boost)
        self.add_emotion(EmotionState.EXCITED, boost * 0.3)
    def on_face_recognized(self, user_name: str):
        """Handle face recognition"""
        self.last_interaction_time = time.time()
        boost = self.personality_config['dynamics']['face_recognition_boost']
        self.add_emotion(EmotionState.HAPPY, boost)
        self.add_emotion(EmotionState.EXCITED, boost * 0.5)
        logger.info(f"Recognized {user_name}")
    def set_emotion_from_llm(self, emotion_str: str, intensity: float = 0.8):
        """
        Set emotion based on LLM's choice
        This is the new primary way to update emotions - the LLM decides the emotion
        Args:
            emotion_str: Emotion name (e.g., "happy", "excited")
            intensity: Emotion intensity (0-1), defaults to 0.8
        """
        self.last_interaction_time = time.time()
        try:
            emotion = EmotionState(emotion_str.lower())
        except ValueError:
            logger.warning(f"Invalid emotion '{emotion_str}', defaulting to happy")
            emotion = EmotionState.HAPPY
        for e in EmotionState:
            if e == emotion:
                self.emotion_scores[e] = intensity
            else:
                self.emotion_scores[e] = max(0.0, self.emotion_scores[e] * 0.3)
        self._update_primary_emotion()
        logger.info(f"Emotion set from LLM: {emotion.value} (intensity: {intensity:.2f})")
    def process_emotion_sequence(self, emotion_list: list):
        """
        Process a sequence of emotions from multi-emotion LLM response
        The final emotion in the sequence becomes dominant, but earlier emotions
        contribute with decreasing weight to create a natural emotional blend.
        Example:
            ["excited", "curious", "happy"]
            ? excited: 0.3, curious: 0.5, happy: 0.8 (final is strongest)
        Args:
            emotion_list: List of emotion strings in sequential order
        """
        if not emotion_list:
            logger.warning("Empty emotion sequence, no update")
            return
        self.last_interaction_time = time.time()
        for e in EmotionState:
            self.emotion_scores[e] = max(0.0, self.emotion_scores[e] * 0.2)
        num_emotions = len(emotion_list)
        for i, emotion_str in enumerate(emotion_list):
            try:
                emotion = EmotionState(emotion_str.lower())
            except ValueError:
                logger.warning(f"Invalid emotion '{emotion_str}' in sequence, skipping")
                continue
            base_intensity = 0.3
            max_intensity = 0.8
            if num_emotions > 1:
                position_weight = i / (num_emotions - 1)
                intensity = base_intensity + (max_intensity - base_intensity) * position_weight
            else:
                intensity = max_intensity
            self.emotion_scores[emotion] = max(
                self.emotion_scores[emotion],
                intensity
            )
        self._update_primary_emotion()
        logger.info(f"Processed emotion sequence: {' ? '.join(emotion_list)} (final: {self.current_e
motion.value})")
    def _update_primary_emotion(self):
        """Update primary emotion based on scores"""
        max_emotion = max(self.emotion_scores.items(), key=lambda x: x[1])
        self.current_emotion = max_emotion[0]
        self.emotion_intensity = max_emotion[1]
    def get_emotion(self) -> str:
        """Get current primary emotion"""
        return self.current_emotion.value
    def get_emotion_data(self) -> Dict:
        """Get full emotion state data"""
        return {
            'emotion': self.current_emotion.value,
            'intensity': self.emotion_intensity,
            'energy': self.energy_level,
            'scores': {e.value: score for e, score in self.emotion_scores.items()}
        }
```

### tests/test_emotion_display.py

```
import pygame
import os
import sys
import glob
SCREEN_SIZE = (320, 240)
IMAGE_DIR = "src/display"
TOGGLE_MS = 1500
def load_emotions(image_dir):
    """
    Automatically load all emotion images
    Naming convention: <emotion>.png (static) and <emotion>_speaking.png (speaking)
    Returns: {emotion_name: (base_surface, speaking_surface)}
    """
    emotions = {}
    base_files = glob.glob(os.path.join(image_dir, "*.png"))
    for base_path in base_files:
        filename = os.path.basename(base_path)
        if "_speaking" in filename or "_active" in filename:
            continue
        emotion_name = os.path.splitext(filename)[0]
        speaking_path = os.path.join(image_dir, f"{emotion_name}_speaking.png")
        base = pygame.image.load(base_path).convert_alpha()
        base = pygame.transform.smoothscale(base, SCREEN_SIZE)
        if os.path.exists(speaking_path):
            speaking = pygame.image.load(speaking_path).convert_alpha()
            speaking = pygame.transform.smoothscale(speaking, SCREEN_SIZE)
        else:
            speaking = base
        emotions[emotion_name] = (base, speaking)
        print(f"Loaded emotion: {emotion_name}")
    return emotions
def load_listening(image_dir):
    """
    Load listening state frames:
      listening.png (base)
      listening_active.png (active, optional - will reuse base if missing)
    Returns: (base_surface, active_surface) or None if not found
    """
    base_path = os.path.join(image_dir, "listening.png")
    active_path = os.path.join(image_dir, "listening_active.png")
    if not os.path.exists(base_path):
        print("Warning: listening.png not found")
        return None
    base = pygame.image.load(base_path).convert_alpha()
    base = pygame.transform.smoothscale(base, SCREEN_SIZE)
    if os.path.exists(active_path):
        active = pygame.image.load(active_path).convert_alpha()
        active = pygame.transform.smoothscale(active, SCREEN_SIZE)
    else:
        active = base
    print("Loaded listening state")
    return (base, active)
def main():
    pygame.init()
    screen = pygame.display.set_mode(SCREEN_SIZE)
    pygame.display.set_caption("Emotion Display Test")
    clock = pygame.time.Clock()
    emotions = load_emotions(IMAGE_DIR)
    listening_frames = load_listening(IMAGE_DIR)
    if not emotions:
        print(f"ERROR: No emotion images found in {IMAGE_DIR}")
        sys.exit(1)
```

```
emotion_list = sorted(emotions.keys())
current_idx = 0
current_emotion = emotion_list[current_idx]
is_speaking = False
is_listening = False
show_alt = False
last_toggle = pygame.time.get_ticks()
print(f"\nLoaded {len(emotions)} emotions: {', '.join(emotion_list)}")
print("\nControls:")
print("  L          - Toggle listening")
print("  SPACE      - Toggle speaking")
print("  LEFT/RIGHT - Switch emotion")
print("  ESC        - Quit")
print(f"\nCurrent: {current_emotion}")
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                running = False
            elif event.key == pygame.K_l:
                is_listening = not is_listening
                show_alt = False
                print(f"Listening: {is_listening}")
            elif event.key == pygame.K_SPACE:
                is_speaking = not is_speaking
                show_alt = False
                print(f"Speaking: {is_speaking}")
            elif event.key == pygame.K_RIGHT:
                current_idx = (current_idx + 1) % len(emotion_list)
                current_emotion = emotion_list[current_idx]
                print(f"Current: {current_emotion}")
                show_alt = False
            elif event.key == pygame.K_LEFT:
                current_idx = (current_idx - 1) % len(emotion_list)
                current_emotion = emotion_list[current_idx]
                print(f"Current: {current_emotion}")
                show_alt = False
    if is_listening or is_speaking:
        now = pygame.time.get_ticks()
        if now - last_toggle >= TOGGLE_MS:
            last_toggle = now
            show_alt = not show_alt
    screen.fill((0, 0, 0))
    if is_listening and listening_frames:
        base, active = listening_frames
        frame = active if show_alt else base
        state_text = "listening"
    elif is_speaking:
        base, speaking = emotions[current_emotion]
        frame = speaking if show_alt else base
        state_text = f"{current_emotion} (speaking)"
    else:
        base, _ = emotions[current_emotion]
        frame = base
        state_text = current_emotion
    screen.blit(frame, (0, 0))
    font = pygame.font.Font(None, 24)
    text = font.render(state_text, True, (255, 255, 0))
    screen.blit(text, (10, 10))
    pygame.display.flip()
    clock.tick(30)
    pygame.quit()
if __name__ == "__main__":
    main()
```

### tests/test_expression_pipeline.py

```
#!/usr/bin/env python3
"""
Expression Pipeline - Standalone Demo
Demonstrates emotion display on piTFT with smooth transitions
Based on two_collide.py reference implementation
"""
import sys
import time
import logging
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
import yaml
from expression import EmotionDisplay
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
def load_config():
    """Load configuration from settings.yaml"""
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        logger.error(f"Config file not found: {config_path}")
        sys.exit(1)
    try:
        with open(config_path) as f:
            config = yaml.safe_load(f)
        logger.info("Configuration loaded")
        return config
    except Exception as e:
        logger.error(f"Failed to load config: {e}")
        sys.exit(1)
def demo_emotion_cycle(display: EmotionDisplay, emotions: list, duration: float = 3.0):
    """
    Cycle through emotions with smooth transitions
    Args:
        display: EmotionDisplay instance
        emotions: List of emotion names to cycle through
        duration: How long to show each emotion (seconds)
    """
    logger.info(f"Starting emotion cycle through {len(emotions)} emotions")
    for emotion in emotions:
        logger.info(f"Showing emotion: {emotion}")
        display.set_emotion(emotion, transition_duration=0.5)
        time.sleep(duration)
def demo_speaking_animation(display: EmotionDisplay):
    """
    Demonstrate speaking animation
    Args:
        display: EmotionDisplay instance
    """
    logger.info("Demonstrating speaking animation")
    display.set_emotion('happy', transition_duration=0.5)
    time.sleep(1.0)
    display.set_speaking(True)
    logger.info("Speaking animation started")
    time.sleep(3.0)
    display.set_speaking(False)
    logger.info("Speaking animation stopped")
    time.sleep(1.0)
def demo_listening_state(display: EmotionDisplay):
    """
    Demonstrate listening state
    Args:
        display: EmotionDisplay instance
    """
    logger.info("Demonstrating listening state")
    display.set_listening(True)
    logger.info("Listening state activated")
    time.sleep(3.0)
    display.set_listening(False)
    logger.info("Listening state deactivated")
    time.sleep(1.0)
def main():
    """Main demo function"""
    print("\n" + "="*70)
    print("? EMOTION EXPRESSION PIPELINE DEMO")
    print("="*70)
    print("\nThis demo cycles through emotions with smooth transitions.")
    print("Press GPIO pin 27 to exit (or Ctrl+C)")
    print("?" * 70)
    config = load_config()
    try:
        display = EmotionDisplay(config)
        logger.info("EmotionDisplay initialized successfully")
    except Exception as e:
        logger.error(f"Failed to initialize display: {e}")
        logger.error("Make sure emotion sprites exist in src/display/")
        return 1
    display.start()
    logger.info("Display started")
    all_emotions = [
        'happy',
        'excited',
        'curious',
        'loving',
        'playful',
        'surprised',
        'bored',
        'sad',
        'lonely',
        'sleepy',
        'scared',
        'angry',
    ]
    try:
        print("\n? Demo 1: Cycling through all 12 emotions...")
        demo_emotion_cycle(display, all_emotions, duration=2.5)
        print("\n??  Demo 2: Speaking animation...")
        demo_speaking_animation(display)
        print("\n? Demo 3: Listening state...")
        demo_listening_state(display)
        print("\n? Demo 4: Rapid emotion changes...")
        rapid_emotions = ['excited', 'happy', 'surprised', 'playful']
        demo_emotion_cycle(display, rapid_emotions, duration=1.5)
        print("\n? Demo 5: Slow smooth transitions...")
        display.set_emotion('happy', transition_duration=2.0)
        time.sleep(3.0)
        display.set_emotion('sad', transition_duration=2.0)
        time.sleep(3.0)
        display.set_emotion('loving', transition_duration=2.0)
        time.sleep(3.0)
        print("\n? Demo complete!")
        print("Press GPIO pin 27 to exit, or waiting 5 seconds...")
        time.sleep(5.0)
    except KeyboardInterrupt:
        print("\n\n??  Demo interrupted by user")
    except Exception as e:
        logger.error(f"Demo error: {e}", exc_info=True)
        return 1
    finally:
        print("\n? Cleaning up...")
        display.cleanup()
        print("? Cleanup complete")
    print("\n" + "="*70)
    print("DEMO SUMMARY")
    print("="*70)
    print("? All emotion transitions demonstrated")
    print("? Speaking animation tested")
    print("? Listening state tested")
    print("\nThe expression pipeline is ready for integration!")
    print("="*70 + "\n")
    return 0
if __name__ == "__main__":
    sys.exit(main())
```

### tests/test_full_conversation.py

```
#!/usr/bin/env python3
"""
Full Conversation Demo
Complete voice-to-voice conversation with LLM
Voice Input ? Whisper STT ? Ollama LLM ? TTS ? Speaker
"""
import sys
import logging
import time
from pathlib import Path
from datetime import datetime
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
import yaml
from llm import ConversationPipeline
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
class FullConversationDemo:
    """Interactive full conversation demo with visual feedback"""
    def __init__(self, config: dict):
        """Initialize demo"""
        self.config = config
        self.pipeline = ConversationPipeline(config)
        self.conversation_log = []
    def start(self):
        """Start the demo"""
        self.print_header()
        if not self._check_prerequisites():
            return False
        self.pipeline.set_callbacks(
            on_listening=self.on_listening,
            on_transcribed=self.on_transcribed,
            on_thinking=self.on_thinking,
            on_responding=self.on_responding,
            on_speaking=self.on_speaking,
            on_complete=self.on_complete
        )
        print("\n? Starting full conversation pipeline...\n")
        self.pipeline.start()
        self.print_instructions()
        return True
    def stop(self):
        """Stop the demo"""
        print("\n\n??  Stopping conversation pipeline...")
        self.pipeline.stop()
        self.print_summary()
    def print_header(self):
        """Print demo header"""
        print("\n" + "=" * 80)
        print("??  FULL CONVERSATION DEMO".center(80))
        print("Voice ? Whisper STT ? Ollama LLM ? TTS ? Speech".center(80))
        print("=" * 80)
    def _check_prerequisites(self) -> bool:
        """Check if all systems are ready"""
        print("\n? Checking prerequisites...")
        checks = []
        print("   ? Testing microphone...")
        mic_ok = self.pipeline.voice_input.test_microphone()
        checks.append(("Microphone", mic_ok))
        print("   ? Testing Ollama...")
        llm_ok = self.pipeline.conversation_manager.llm.is_available
        checks.append(("Ollama LLM", llm_ok))
        print("   ? Testing TTS...")
        try:
            self.pipeline.tts.speak("Test", wait=False)
            tts_ok = True
        except:
            tts_ok = False
        checks.append(("Text-to-Speech", tts_ok))
        print("\n   Results:")
        all_ok = True
        for name, ok in checks:
            status = "?" if ok else "?"
            print(f"      {status} {name}")
            if not ok:
                all_ok = False
        if not all_ok:
            print("\n   ? Some components failed - see troubleshooting below")
            self.print_troubleshooting()
            return False
        print("\n   ? All systems ready!")
        return True
    def print_instructions(self):
        """Print usage instructions"""
        print("?" + "?" * 78 + "?")
        print("?" + " HOW TO USE ".center(78) + "?")
        print("?" + "?" * 78 + "?")
        print("? 1. Speak naturally into your microphone                                   ?")
        print("? 2. Wait for the bot to detect speech end (automatic)                      ?")
        print("? 3. System will transcribe, think, and respond                             ?")
        print("? 4. Listen to the bot's voice response                                     ?")
        print("? 5. Continue the conversation!                                             ?")
        print("?                                                                            ?")
        print("? Press Ctrl+C to stop and see statistics                                   ?")
        print("?" + "?" * 78 + "?")
        print("\n? Ready! Start speaking...\n")
    def print_troubleshooting(self):
        """Print troubleshooting tips"""
        print("\n" + "=" * 80)
        print("? TROUBLESHOOTING")
        print("=" * 80)
        print("\nMicrophone issues:")
        print("  ? Check connection: arecord -l")
        print("  ? Test recording: arecord -D hw:1,0 -d 3 test.wav && aplay test.wav")
        print("  ? Adjust volume: alsamixer")
        print("\nOllama issues:")
        print("  ? Start service: ollama serve")
        print(f"  ? Pull model: ollama pull {self.config['llm']['ollama']['model']}")
        print("  ? Check running: curl http://localhost:11434/api/tags")
        print("\nTTS issues:")
        print("  ? Check speaker: aplay /usr/share/sounds/alsa/Front_Center.wav")
        print("  ? Install espeak: sudo apt-get install espeak")
        print("=" * 80)
    def on_listening(self):
        """Called when user starts speaking"""
        timestamp = datetime.now().strftime("%H:%M:%S")
        print(f"\n[{timestamp}] ? LISTENING...")
        print("?")
    def on_transcribed(self, text: str):
        """Called when speech is transcribed"""
        print("?")
        print("?? ? YOU SAID:")
        print(f"   \"{text}\"\n")
        self.conversation_log.append({
            'timestamp': datetime.now(),
            'role': 'user',
            'text': text
        })
    def on_thinking(self):
        """Called when LLM is processing"""
        print("   ? Thinking...")
    def on_responding(self, text: str, emotion: str):
        """Called when response is ready"""
        print(f"   ? Emotion: {emotion}")
        print(f"   ? BOT SAYS:")
        print(f"   \"{text}\"")
        self.conversation_log.append({
            'timestamp': datetime.now(),
            'role': 'bot',
            'text': text,
            'emotion': emotion
        })
    def on_speaking(self):
        """Called when TTS starts"""
        print("   ? Speaking...\n")
    def on_complete(self):
        """Called when full cycle completes"""
        print("?" * 80)
    def print_summary(self):
        """Print session summary"""
        stats = self.pipeline.get_statistics()
        print("\n" + "=" * 80)
        print("? SESSION SUMMARY".center(80))
        print("=" * 80)
        print(f"\n? Conversation Statistics:")
        print(f"   Total conversations: {stats['conversations']}")
        print(f"   Avg response time: {stats['avg_response_time']:.2f}s")
        print(f"\n? Voice Input:")
        voice_stats = stats['voice_input']
        print(f"   Total utterances: {voice_stats['total_utterances']}")
        print(f"   Avg transcription time: {voice_stats['avg_transcription_time']:.2f}s")
        print(f"\n? LLM:")
        llm_stats = stats['llm']
        print(f"   Total requests: {llm_stats['total_requests']}")
        print(f"   Total tokens: {llm_stats['total_tokens']}")
        print(f"   Avg time: {llm_stats['avg_time_per_request']:.2f}s")
        print(f"\n? TTS:")
        tts_stats = stats['tts']
        print(f"   Total utterances: {tts_stats['total_utterances']}")
        print(f"\n? Current State:")
        conv_stats = stats['conversation']
        print(f"   Emotion: {conv_stats['current_emotion']}")
        print(f"   Messages: {conv_stats['message_count']}")
        if self.conversation_log:
            print(f"\n? Conversation Log:")
            for i, entry in enumerate(self.conversation_log, 1):
                time_str = entry['timestamp'].strftime("%H:%M:%S")
                role_icon = "?" if entry['role'] == 'user' else "?"
                emotion_str = f" ({entry['emotion']})" if 'emotion' in entry else ""
                print(f"   [{time_str}] {role_icon} {entry['text'][:60]}{emotion_str}")
        print("\n" + "=" * 80)
        print("? Demo completed successfully!".center(80))
        print("=" * 80)
    def run(self):
        """Run the demo"""
        if not self.start():
            return 1
        try:
            while True:
                time.sleep(0.1)
        except KeyboardInterrupt:
            pass
        finally:
            self.stop()
            self.pipeline.cleanup()
        return 0
def main():
    """Main entry point"""
    print("\n" + "=" * 80)
    print("? FULL CONVERSATION DEMO - Voice-to-Voice AI")
    print("=" * 80)
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print("? Config file not found!")
        print(f"   Looking for: {config_path}")
        return 1
    try:
        with open(config_path) as f:
            config = yaml.safe_load(f)
        print("? Configuration loaded")
    except Exception as e:
        print(f"? Failed to load config: {e}")
        return 1
    demo = FullConversationDemo(config)
    return demo.run()
if __name__ == "__main__":
    sys.exit(main())
```

### tests/test_llm_only.py

```
#!/usr/bin/env python3
"""
LLM Integration Test Script
Test Ollama integration and conversation without voice
"""
import sys
import logging
import time
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
import yaml
from llm import OllamaClient, ConversationManager, TTSEngine
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
def test_ollama_connection(config):
    """Test basic Ollama connection"""
    print("\n" + "="*70)
    print("TEST 1: Ollama Connection")
    print("="*70)
    client = OllamaClient(config)
    if client.is_available:
        print("? Ollama is running and available")
        info = client.get_model_info()
        if info:
            print(f"   Model: {info.get('model', 'Unknown')}")
            print(f"   Size: {info.get('size', 0) / 1e9:.2f} GB")
        return True
    else:
        print("? Ollama is NOT available")
        print("\n   To fix:")
        print("   1. Start Ollama: ollama serve")
        print(f"   2. Pull model: ollama pull {config['llm']['ollama']['model']}")
        return False
def test_text_generation(config):
    """Test basic text generation"""
    print("\n" + "="*70)
    print("TEST 2: Text Generation")
    print("="*70)
    client = OllamaClient(config)
    if not client.is_available:
        print("??  Skipping (Ollama not available)")
        return False
    print("\nGenerating response...")
    result = client.generate("Say hello in one short sentence!")
    print(f"\n? Response: {result['response']}")
    print(f"??  Time: {result['duration']:.2f}s")
    print(f"? Tokens: {result['tokens']}")
    return True
def test_personality_prompts(config):
    """Test personality-aware generation"""
    print("\n" + "="*70)
    print("TEST 3: Personality Prompts")
    print("="*70)
    client = OllamaClient(config)
    if not client.is_available:
        print("??  Skipping (Ollama not available)")
        return False
    test_prompts = [
        ("Hello! How are you today?", "should express greeting emotion"),
        ("I just won a prize!", "should express excited/happy emotion"),
        ("Tell me something interesting", "should express curious emotion"),
    ]
    for prompt, expected_behavior in test_prompts:
        print(f"\n? Testing: '{prompt}'")
        print(f"   Expected: {expected_behavior}")
        result = client.generate_with_personality(
            user_input=prompt,
            user_name="friend"
        )
        response = result['response']
        print(f"   Response: {response}")
        if response.startswith('[') and ']' in response:
            emotion_end = response.index(']')
            emotion = response[1:emotion_end]
            print(f"   ? LLM chose emotion: {emotion}")
        else:
            print("   ??  No emotion tag detected in response")
    return True
def test_conversation_manager(config):
    """Test conversation manager with context"""
    print("\n" + "="*70)
    print("TEST 4: Conversation Manager")
    print("="*70)
    manager = ConversationManager(config)
    if not manager.llm.is_available:
        print("??  Skipping (Ollama not available)")
        return False
    conversation = [
        "Hello! What's your name?",
        "Nice to meet you! How are you feeling?",
        "What do you like to do for fun?"
    ]
    print("\nHaving a conversation...\n")
    for i, msg in enumerate(conversation, 1):
        print(f"[{i}] ? You: {msg}")
        response, metadata = manager.process_user_input(msg)
        print(f"    ? Bot ({metadata['emotion']}): {response}")
        print(f"    ??  {metadata['response_time']:.2f}s | "
              f"? {metadata['emotion']} | "
              f"? {metadata['energy']:.0%}\n")
        time.sleep(0.5)
    print("?" * 70)
    print(manager.get_conversation_summary())
    return True
def test_tts_engine(config):
    """Test TTS with emotions"""
    print("\n" + "="*70)
    print("TEST 5: TTS Engine")
    print("="*70)
    tts = TTSEngine(config)
    print("\nAvailable voices:")
    voices = tts.get_available_voices()
    for voice in voices[:3]:
        print(f"   [{voice['index']}] {voice['name']}")
    print("\nTesting TTS with emotions...")
    test_phrase = "Hello! This is a test."
    emotions_to_test = ['happy', 'sad', 'excited']
    for emotion in emotions_to_test:
        print(f"\n   ? Speaking with '{emotion}' emotion...")
        tts.speak(test_phrase, emotion=emotion, wait=True)
        time.sleep(0.3)
    tts.cleanup()
    print("\n? TTS test complete")
    return True
def interactive_mode(config):
    """Interactive conversation mode"""
    print("\n" + "="*70)
    print("INTERACTIVE MODE")
    print("="*70)
    print("\nType your messages (or 'quit' to exit)")
    print("?" * 70)
    manager = ConversationManager(config)
    if not manager.llm.is_available:
        print("? Ollama not available - cannot run interactive mode")
        return False
    conversation_num = 0
    try:
        while True:
            user_input = input("\n? You: ").strip()
            if not user_input:
                continue
            if user_input.lower() in ['quit', 'exit', 'bye']:
                print("\n? Goodbye!")
                break
            conversation_num += 1
            response, metadata = manager.process_user_input(user_input)
            print(f"? Bot ({metadata['emotion']}): {response}")
            if conversation_num % 3 == 0:
                print(f"    [??  {metadata['response_time']:.2f}s | "
                      f"? {metadata['emotion']} | "
                      f"? {metadata['tokens']} tokens]")
    except KeyboardInterrupt:
        print("\n\n? Goodbye!")
    print("\n" + "?" * 70)
    print(manager.get_conversation_summary())
    return True
def test_emotion_segments(config):
    """Test that metadata includes emotion_segments for multi-emotion responses"""
    print("\n" + "="*70)
    print("TEST 6: Emotion Segments in Metadata")
    print("="*70)
    manager = ConversationManager(config)
    if not manager.llm.is_available:
        print("??  Skipping (Ollama not available)")
        return False
    print("\n? Testing emotion segments...")
    response, metadata = manager.process_user_input(
        "Tell me a short story with different emotions!"
    )
    print(f"\n? Response: {response}")
    assert 'emotion_segments' in metadata, "metadata should include emotion_segments"
    assert isinstance(metadata['emotion_segments'], list), "emotion_segments should be a list"
    segments = metadata['emotion_segments']
    print(f"\n? Found {len(segments)} emotion segment(s)")
    for i, segment in enumerate(segments, 1):
        assert isinstance(segment, tuple), f"Segment {i} should be a tuple"
        assert len(segment) == 2, f"Segment {i} should have 2 elements (emotion, text)"
        emotion, text = segment
        assert isinstance(emotion, str), f"Segment {i} emotion should be a string"
        assert isinstance(text, str), f"Segment {i} text should be a string"
        print(f"   Segment {i}: ({emotion}) {text[:50]}...")
    print("\n? Emotion segments test complete")
    return True
def test_segmented_tts(config):
    """Test segmented TTS with multiple emotions"""
    print("\n" + "="*70)
    print("TEST 7: Segmented TTS")
    print("="*70)
    tts = TTSEngine(config)
    segments = [
        ("excited", "Hello friend!"),
        ("curious", "What are you up to today?"),
        ("happy", "That sounds great!")
    ]
    print(f"\n? Testing segmented speech with {len(segments)} segments...")
    for i, (emotion, text) in enumerate(segments, 1):
        print(f"   Segment {i}: ({emotion}) {text}")
    tts.speak_segments_with_emotions(segments, wait=True)
    print("\n? Segmented TTS test complete")
    return True
def main():
    """Main test function"""
    print("\n" + "="*70)
    print("? LLM INTEGRATION TEST SUITE")
    print("="*70)
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print("? Config file not found!")
        print(f"   Looking for: {config_path}")
        return 1
    try:
        with open(config_path) as f:
            config = yaml.safe_load(f)
        print("? Configuration loaded")
    except Exception as e:
        print(f"? Failed to load config: {e}")
        return 1
    tests = [
        ("Ollama Connection", test_ollama_connection),
        ("Text Generation", test_text_generation),
        ("Personality Prompts", test_personality_prompts),
        ("Conversation Manager", test_conversation_manager),
        ("TTS Engine", test_tts_engine),
        ("Emotion Segments", test_emotion_segments),
        ("Segmented TTS", test_segmented_tts),
    ]
    results = []
    for test_name, test_func in tests:
        try:
            result = test_func(config)
            results.append((test_name, result))
        except KeyboardInterrupt:
            print("\n\n??  Tests interrupted")
            return 1
        except Exception as e:
            print(f"\n? Test failed with error: {e}")
            logger.error("Test error", exc_info=True)
            results.append((test_name, False))
    print("\n" + "="*70)
    print("TEST SUMMARY")
    print("="*70)
    passed = sum(1 for _, result in results if result)
    total = len(results)
    for test_name, result in results:
        status = "? PASS" if result else "? FAIL/SKIP"
        print(f"{status} - {test_name}")
    print(f"\nTotal: {passed}/{total} tests passed")
    if passed > 0:
        print("\n" + "="*70)
        response = input("\nRun interactive mode? (y/n): ").strip().lower()
        if response == 'y':
            interactive_mode(config)
    print("\n? All tests complete!")
    return 0
if __name__ == "__main__":
    sys.exit(main())
```

### tests/test_memory_system.py

```
#!/usr/bin/env python3
"""
Memory System Test Script
Test user profiles, conversation persistence, and database functionality
"""
import sys
import logging
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
import yaml
from memory import initialize_memory
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
def test_user_profiles(user_memory):
    """Test user profile management"""
    print("\n" + "="*70)
    print("TEST 1: User Profile Management")
    print("="*70)
    print("\n? Creating test users...")
    john_id = user_memory.create_user("John")
    alice_id = user_memory.create_user("Alice")
    print(f"   Created: John (ID: {john_id}), Alice (ID: {alice_id})")
    print("\n? Getting user by ID...")
    john = user_memory.get_user_by_id(john_id)
    print(f"   User {john_id}: {john['name']}")
    print("\n? Getting user by name...")
    alice = user_memory.get_user_by_name("Alice")
    print(f"   Found: {alice['name']} (ID: {alice['user_id']})")
    print("\n? All users:")
    all_users = user_memory.get_all_users()
    for user in all_users:
        print(f"   ID {user['user_id']}: {user['name']}")
    print("\n? User profile tests passed")
    return john_id, alice_id
def test_preferences(user_memory, user_id):
    """Test user preferences"""
    print("\n" + "="*70)
    print("TEST 2: User Preferences")
    print("="*70)
    print("\n? Setting preferences...")
    user_memory.set_preference(user_id, "favorite_color", "blue")
    user_memory.set_preference(user_id, "hobby", "reading")
    print("   Set favorite_color=blue")
    print("   Set hobby=reading")
    print("\n? Getting preference...")
    color = user_memory.get_preference(user_id, "favorite_color")
    print(f"   favorite_color: {color}")
    print("\n? All preferences:")
    all_prefs = user_memory.get_all_preferences(user_id)
    for key, value in all_prefs.items():
        print(f"   {key}: {value}")
    print("\n? Preference tests passed")
def test_interactions(user_memory, user_id):
    """Test interaction logging"""
    print("\n" + "="*70)
    print("TEST 3: Interaction Logging")
    print("="*70)
    print("\n? Recording interactions...")
    user_memory.record_interaction(user_id, "voice", "Hello!", "happy")
    user_memory.record_interaction(user_id, "touch", "head", "excited")
    user_memory.record_interaction(user_id, "voice", "How are you?", "curious")
    print("   Recorded 3 interactions")
    print("\n? Interaction history:")
    history = user_memory.get_interaction_history(user_id, limit=10)
    for interaction in history:
        print(f"   [{interaction['timestamp']}] {interaction['interaction_type']}: "
              f"{interaction['interaction_value']} ? {interaction['emotion_response']}")
    print("\n? Interaction stats:")
    stats = user_memory.get_interaction_stats(user_id)
    for interaction_type, count in stats.items():
        print(f"   {interaction_type}: {count}")
    print("\n? Interaction tests passed")
def test_conversation_persistence(conversation_history, user_id):
    """Test conversation persistence"""
    print("\n" + "="*70)
    print("TEST 4: Conversation Persistence")
    print("="*70)
    session_id = conversation_history.generate_session_id()
    print(f"\n? Session ID: {session_id}")
    print("\n? Saving conversation...")
    conversation_history.save_message(user_id, session_id, "user", "Hello!")
    conversation_history.save_message(user_id, session_id, "assistant", "Hi! How are you?", emotion=
"happy", tokens=5)
    conversation_history.save_message(user_id, session_id, "user", "I'm great, thanks!")
    conversation_history.save_message(user_id, session_id, "assistant", "That's wonderful!", emotion
="excited", tokens=3)
    print("   Saved 4 messages")
    print("\n? Session conversation:")
    messages = conversation_history.get_session_conversation(session_id)
    for msg in messages:
        emotion_str = f" ({msg['emotion']})" if msg['emotion'] else ""
        print(f"   {msg['role']}: {msg['message']}{emotion_str}")
    print("\n? User's recent conversations:")
    recent = conversation_history.get_user_conversations(user_id, limit=5)
    for msg in recent:
        print(f"   [{msg['session_id'][:8]}...] {msg['role']}: {msg['message'][:40]}...")
    print("\n? Conversation stats:")
    stats = conversation_history.get_conversation_stats(user_id)
    print(f"   Total messages: {stats['total_messages']}")
    print(f"   Total sessions: {stats['total_sessions']}")
    print(f"   Avg messages/session: {stats['avg_messages_per_session']:.1f}")
    if stats.get('top_emotions'):
        print(f"   Top emotions: {stats['top_emotions']}")
    print("\n? Conversation persistence tests passed")
    return session_id
def test_search(conversation_history):
    """Test conversation search"""
    print("\n" + "="*70)
    print("TEST 5: Conversation Search")
    print("="*70)
    print("\n? Searching for 'great'...")
    results = conversation_history.search_conversations("great")
    print(f"   Found {len(results)} matches:")
    for result in results[:3]:
        print(f"   {result['role']}: {result['message']}")
    print("\n? Search tests passed")
def test_memory_persistence():
    """Test that memory persists across instances"""
    print("\n" + "="*70)
    print("TEST 6: Memory Persistence")
    print("="*70)
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    with open(config_path) as f:
        config = yaml.safe_load(f)
    print("\n? Creating first memory instance...")
    user_memory1, conv_history1 = initialize_memory(config)
    test_name = "MemoryTestUser"
    test_user_id = user_memory1.create_user(test_name)
    user_memory1.set_preference(test_user_id, "test_key", "test_value")
    print(f"   Created user: {test_name} (ID: {test_user_id})")
    print("\n? Creating second memory instance (simulating restart)...")
    user_memory2, conv_history2 = initialize_memory(config)
    print("\n? Checking if data persists...")
    retrieved_user = user_memory2.get_user_by_name(test_name)
    if retrieved_user:
        print(f"   ? User found: {retrieved_user['name']} (ID: {retrieved_user['user_id']})")
        pref_value = user_memory2.get_preference(retrieved_user['user_id'], "test_key")
        if pref_value == "test_value":
            print(f"   ? Preference persisted: test_key={pref_value}")
        else:
            print(f"   ? Preference not found")
    else:
        print(f"   ? User not found after restart")
    print("\n? Persistence tests passed")
def test_cleanup(conversation_history):
    """Test cleanup functionality"""
    print("\n" + "="*70)
    print("TEST 7: Database Cleanup")
    print("="*70)
    print("\n??  Testing cleanup (0 days - should delete nothing recent)...")
    deleted = conversation_history.cleanup_old_conversations(days=0)
    print(f"   Deleted {deleted} old conversations")
    print("\n? Cleanup tests passed")
def main():
    """Main test function"""
    print("\n" + "="*70)
    print("? MEMORY SYSTEM TEST SUITE")
    print("="*70)
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print("? Config file not found!")
        print(f"   Looking for: {config_path}")
        return 1
    try:
        with open(config_path) as f:
            config = yaml.safe_load(f)
        print("? Configuration loaded")
    except Exception as e:
        print(f"? Failed to load config: {e}")
        return 1
    try:
        print("\n? Initializing memory system...")
        user_memory, conversation_history = initialize_memory(config)
        print("? Memory system initialized")
    except Exception as e:
        print(f"? Failed to initialize memory: {e}")
        logger.error("Initialization error", exc_info=True)
        return 1
    try:
        john_id, alice_id = test_user_profiles(user_memory)
        test_preferences(user_memory, john_id)
        test_interactions(user_memory, john_id)
        session_id = test_conversation_persistence(conversation_history, alice_id)
        test_search(conversation_history)
        test_cleanup(conversation_history)
        test_memory_persistence()
    except KeyboardInterrupt:
```

```
        print("\n\n??  Tests interrupted")
        return 1
    except Exception as e:
        print(f"\n? Test failed with error: {e}")
        logger.error("Test error", exc_info=True)
        return 1
    print("\n" + "="*70)
    print("TEST SUMMARY")
    print("="*70)
    print("? All memory system tests passed!")
    print("\n? Database location:", config.get('memory', {}).get('database_path', 'data/companion.db
'))
    return 0
if __name__ == "__main__":
    sys.exit(main())
```

### tests/test_ollama_direct.sh

```bash
#!/bin/bash
echo "=== Testing Ollama Directly ==="
echo ""
echo "1. Testing simple prompt:"
ollama run qwen2.5:0.5b "Say hello in one sentence"
echo ""
echo "2. Testing with emotion tag:"
ollama run qwen2.5:0.5b "You are Buddy. Respond with [happy] tag at start. Say hello."
echo ""
echo "3. Testing with short system prompt:"
ollama run qwen2.5:0.5b "You are Buddy, a pet robot. Format: [emotion] message. Say hello."
echo ""
echo "=== Check Ollama Status ==="
ollama ps
echo ""
echo "=== Check Ollama Logs (last 20 lines) ==="
echo "Run: journalctl -u ollama -n 20"
echo "Or check: ~/.ollama/logs/"
```

### tests/test_tts_hardware.py

```
#!/usr/bin/env python3
"""
TTS Hardware Test Script
Tests TTS output with wm8960-soundcard and all emotion voices
"""
import sys
import logging
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
import yaml
from llm.tts_engine import TTSEngine
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
def test_audio_device():
    """Display audio device information"""
    print("\n" + "="*70)
    print("AUDIO DEVICE INFORMATION")
    print("="*70)
    try:
        import subprocess
        print("\n? ALSA Playback Devices:")
        result = subprocess.run(['aplay', '-l'], capture_output=True, text=True)
        if result.returncode == 0:
            print(result.stdout)
        else:
            print("   ??  Could not list ALSA devices")
        print("\n? Testing default audio device:")
        print("   Playing a brief test tone...")
        result = subprocess.run(
            ['speaker-test', '-t', 'wav', '-c', '2', '-l', '1'],
            capture_output=True,
            text=True,
            timeout=5
        )
        if result.returncode == 0:
            print("   ? Audio playback successful!")
        else:
            print("   ? Audio test failed")
            print(result.stderr)
    except FileNotFoundError:
        print("   ??  ALSA tools not found (install with: sudo apt install alsa-utils)")
    except Exception as e:
        print(f"   ??  Error checking audio: {e}")
def test_tts_engine(config):
    """Test TTS engine with basic speech"""
    print("\n" + "="*70)
    print("TTS ENGINE TEST")
    print("="*70)
    try:
        print("\n? Initializing TTS engine...")
        tts = TTSEngine(config)
        print("   ? TTS engine initialized")
        print("\n??  TTS Configuration:")
        print(f"   Provider: {config.get('speech', {}).get('tts', {}).get('provider', 'pyttsx3')}")
        tts_config = config.get('speech', {}).get('tts', {}).get('pyttsx3', {})
        print(f"   Rate: {tts_config.get('rate', 150)} words/min")
        print(f"   Volume: {tts_config.get('volume', 0.9)}")
        print(f"   Pitch: {tts_config.get('pitch', 1.5)}")
        print("\n? Testing basic speech:")
        print("   Speaking: 'Hello! Audio test.'")
        tts.speak("Hello! Audio test.", wait=True)
        print("   ? Basic speech complete")
        return tts
    except Exception as e:
        print(f"   ? TTS engine failed: {e}")
        logger.error("TTS initialization error", exc_info=True)
        return None
def test_emotion_voices(tts):
    """Test all emotion voices"""
    print("\n" + "="*70)
    print("EMOTION VOICE TEST")
    print("="*70)
    print("\nTesting all 12 emotion states...")
    print("(Listen for rate, pitch, and volume changes)")
    print("")
    emotions_to_test = [
        ('happy', "I'm so happy to see you!"),
        ('excited', "This is so exciting!"),
        ('sad', "I feel sad when you're away."),
        ('sleepy', "I'm feeling so sleepy..."),
        ('angry', "I'm angry about this!"),
        ('scared', "That was scary!"),
        ('loving', "I love you so much!"),
        ('playful', "Let's play together!"),
        ('curious', "I wonder what that is?"),
        ('lonely', "I'm feeling lonely."),
        ('bored', "This is boring."),
        ('surprised', "Wow! What a surprise!"),
    ]
    for i, (emotion, text) in enumerate(emotions_to_test, 1):
        print(f"[{i}/12] {emotion.upper():12s} - {text}")
        try:
            tts.speak_with_emotion(text, emotion, wait=True)
        except Exception as e:
            print(f"        ? Error: {e}")
    print("\n? All emotion voices tested")
def test_multi_emotion_speech(tts):
    """Test multi-emotion speech with transitions"""
    print("\n" + "="*70)
    print("MULTI-EMOTION SPEECH TEST")
    print("="*70)
    print("\nTesting emotion transitions within a single response...")
    print("")
    segments = [
        ('happy', "Hi there! I'm glad to see you!"),
        ('curious', "What have you been up to?"),
        ('excited', "I can't wait to hear about it!"),
    ]
    print("Segments:")
    for emotion, text in segments:
        print(f"  [{emotion}] {text}")
    print("\n? Speaking with emotion transitions...")
    try:
        tts.speak_segments_with_emotions(segments, wait=True)
        print("? Multi-emotion speech complete")
    except Exception as e:
        print(f"? Multi-emotion speech failed: {e}")
        logger.error("Multi-emotion speech error", exc_info=True)
def test_statistics(tts):
    """Display TTS statistics"""
    print("\n" + "="*70)
    print("TTS STATISTICS")
    print("="*70)
    stats = tts.get_statistics()
    print(f"\n? Performance Stats:")
    print(f"   Total utterances: {stats['total_utterances']}")
    print(f"   Total duration: {stats['total_duration']:.2f}s")
    print(f"   Current emotion: {stats['current_emotion'] or 'neutral'}")
    print(f"   Provider: {stats['provider']}")
def main():
    """Main test function"""
    print("\n" + "="*70)
    print("? TTS HARDWARE TEST - wm8960-soundcard")
    print("="*70)
    print("\nThis script tests TTS output with your audio hardware.")
    print("Make sure speakers are connected to the wm8960-soundcard.")
    print("")
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print("? Config file not found!")
        print(f"   Looking for: {config_path}")
        return 1
    try:
        with open(config_path) as f:
            config = yaml.safe_load(f)
        print("? Configuration loaded")
    except Exception as e:
        print(f"? Failed to load config: {e}")
        return 1
    try:
        test_audio_device()
        tts = test_tts_engine(config)
        if not tts:
            print("\n? Cannot continue without TTS engine")
            return 1
        test_emotion_voices(tts)
        test_multi_emotion_speech(tts)
        test_statistics(tts)
        print("\n" + "="*70)
        print("CLEANUP")
        print("="*70)
        tts.cleanup()
        print("? TTS engine cleanup complete")
    except KeyboardInterrupt:
        print("\n\n??  Test interrupted by user")
        return 1
    except Exception as e:
        print(f"\n? Test failed with error: {e}")
        logger.error("Test error", exc_info=True)
        return 1
    print("\n" + "="*70)
    print("TEST SUMMARY")
    print("="*70)
    print("\n? All TTS hardware tests complete!")
    print("\nIf you heard all the test phrases, your audio setup is working correctly.")
    print("The companion bot will now be able to speak using the wm8960-soundcard.")
    return 0
if __name__ == "__main__":
    sys.exit(main())
```

### tests/test_voice_input.py

```
#!/usr/bin/env python3
"""
Voice Input Test Script
Test mini microphone with Whisper STT
"""
import sys
import logging
import time
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
import yaml
from llm.voice_pipeline import VoicePipeline
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
def main():
    """Main test function"""
    print("=" * 70)
    print("? Voice Input Test - Mini Microphone + Whisper")
    print("=" * 70)
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print("? Config file not found at:", config_path)
        print("Run setup.sh first!")
        return 1
    print("\n? Loading configuration...")
    with open(config_path) as f:
        config = yaml.safe_load(f)
    print("\n? Audio Configuration:")
    print(f"  Sample Rate: {config['audio']['input']['sample_rate']} Hz")
    print(f"  Channels: {config['audio']['input']['channels']}")
    print(f"  Chunk Size: {config['audio']['input']['chunk_size']}")
    print(f"  VAD Aggressiveness: {config['audio']['processing']['vad_aggressiveness']}")
    print("\n? Whisper Configuration:")
    print(f"  Model: {config['speech']['stt']['whisper']['model_size']}")
    print(f"  Device: {config['speech']['stt']['whisper']['device']}")
    print(f"  Language: {config['speech']['stt']['language']}")
    print("\n? Initializing voice pipeline...")
    try:
        pipeline = VoicePipeline(config)
    except Exception as e:
        print(f"? Failed to initialize pipeline: {e}")
        return 1
    print("\n??  Testing microphone...")
    if not pipeline.test_microphone():
        print("? Microphone test failed!")
        print("\nTroubleshooting:")
        print("  1. Check if mini microphone is plugged in")
        print("  2. Run: arecord -l")
        print("  3. Test with: arecord -D hw:1,0 -d 3 test.wav && aplay test.wav")
        return 1
    print("? Microphone working!")
    transcription_count = 0
    def on_transcription(result):
        nonlocal transcription_count
        transcription_count += 1
        print("\n" + "=" * 70)
        print(f"? TRANSCRIPTION #{transcription_count}")
        print("=" * 70)
        print(f"Text:       {result['text']}")
        print(f"Language:   {result['language']}")
        print(f"Confidence: {result['confidence']:.2%}")
        print(f"Duration:   {result['duration']:.2f}s")
        print("=" * 70)
    def on_speech_start():
        print("\n? [LISTENING] Speak now...")
    def on_speech_end():
        print("? [PROCESSING] Transcribing with Whisper...")
    pipeline.set_transcription_callback(on_transcription)
    pipeline.set_speech_callbacks(on_speech_start, on_speech_end)
    print("\n? Voice pipeline ready!")
    print("\n" + "=" * 70)
    print("Instructions:")
    print("  1. Speak clearly into your mini microphone")
    print("  2. The system will detect when you start/stop speaking")
    print("  3. Wait for transcription results")
    print("  4. Press Ctrl+C to stop")
    print("=" * 70)
    print("\n? Starting voice recognition...\n")
    try:
        pipeline.start()
        while True:
            time.sleep(0.1)
    except KeyboardInterrupt:
        print("\n\n??  Stopping voice pipeline...")
    finally:
        pipeline.cleanup()
        stats = pipeline.get_statistics()
        print("\n" + "=" * 70)
        print("? Session Statistics")
        print("=" * 70)
        print(f"Total Utterances:         {stats['total_utterances']}")
        print(f"Total Transcription Time: {stats['total_transcription_time']:.2f}s")
        print(f"Avg Time per Utterance:   {stats['avg_transcription_time']:.2f}s")
        print(f"Last Transcription Time:  {stats['last_transcription_time']:.2f}s")
        print("=" * 70)
        if stats['total_utterances'] > 0:
            print("\n? Test completed successfully!")
            print(f"   Transcribed {stats['total_utterances']} utterance(s)")
        else:
            print("\n??  No speech detected")
            print("   Try speaking louder or closer to the microphone")
    return 0
if __name__ == "__main__":
    sys.exit(main())
```

### tests/test_voice_input_simu_response.py

```
#!/usr/bin/env python3
"""
Voice Assistant Demo
Complete demo of voice input with mini microphone and Whisper
Shows real-time transcription with visual feedback
"""
import sys
import logging
import time
from pathlib import Path
from datetime import datetime
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
import yaml
from llm.voice_pipeline import VoicePipeline
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
class VoiceAssistantDemo:
    """Interactive voice assistant demo"""
    def __init__(self, config: dict):
        """Initialize demo"""
        self.config = config
        self.pipeline = VoicePipeline(config)
        self.is_running = False
        self.conversation_history = []
    def start(self):
        """Start the demo"""
        self.print_header()
        print("\n??  Testing microphone...")
        if not self.pipeline.test_microphone():
            print("? Microphone test failed!")
            self.print_troubleshooting()
            return False
        print("? Microphone is working!\n")
        self.pipeline.set_transcription_callback(self.on_transcription)
        self.pipeline.set_speech_callbacks(self.on_speech_start, self.on_speech_end)
        print("? Starting voice assistant...\n")
        self.pipeline.start()
        self.is_running = True
        self.print_instructions()
        return True
    def stop(self):
        """Stop the demo"""
        if not self.is_running:
            return
        print("\n\n??  Stopping voice assistant...")
        self.is_running = False
        self.pipeline.cleanup()
        self.print_summary()
    def print_header(self):
        """Print demo header"""
        print("\n" + "=" * 80)
        print("? VOICE ASSISTANT DEMO".center(80))
        print("Mini Microphone + OpenAI Whisper".center(80))
        print("=" * 80)
    def print_instructions(self):
        """Print usage instructions"""
        print("?" + "?" * 78 + "?")
        print("?" + " INSTRUCTIONS ".center(78) + "?")
        print("?" + "?" * 78 + "?")
        print("? 1. Speak clearly into your mini microphone                               ?")
        print("? 2. The system will automatically detect when you start/stop speaking      ?")
        print("? 3. Wait for transcription results                                         ?")
        print("? 4. Try different commands and questions                                   ?")
        print("? 5. Press Ctrl+C to stop and see statistics                                ?")
        print("?" + "?" * 78 + "?")
        print("\n? Ready! Start speaking...\n")
    def print_troubleshooting(self):
        """Print troubleshooting tips"""
        print("\n" + "=" * 80)
        print("Troubleshooting Tips:")
        print("=" * 80)
        print("1. Check microphone connection:")
        print("   arecord -l")
        print("\n2. Test recording:")
        print("   arecord -D hw:1,0 -d 3 test.wav && aplay test.wav")
        print("\n3. Adjust volume:")
        print("   alsamixer")
        print("\n4. Check config:")
        print("   config/settings.yaml")
        print("=" * 80)
    def on_speech_start(self):
        """Called when speech is detected"""
        timestamp = datetime.now().strftime("%H:%M:%S")
        print(f"\n[{timestamp}] ? LISTENING...")
        print("?")
    def on_speech_end(self):
        """Called when speech ends"""
        print("?")
        print("?? ? Processing with Whisper...")
    def on_transcription(self, result: dict):
        """Called when transcription is complete"""
        text = result['text']
        confidence = result['confidence']
        language = result['language']
        duration = result['duration']
        self.conversation_history.append({
            'timestamp': datetime.now(),
            'text': text,
            'confidence': confidence,
            'language': language,
            'duration': duration
        })
        self.print_transcription_result(result)
        self.simulate_response(text)
    def print_transcription_result(self, result: dict):
        """Print formatted transcription result"""
        text = result['text']
        confidence = result['confidence']
        language = result['language']
        duration = result['duration']
        if confidence >= 0.8:
            conf_emoji = "?"
            conf_level = "HIGH"
        elif confidence >= 0.5:
            conf_emoji = "??"
            conf_level = "MEDIUM"
        else:
            conf_emoji = "?"
            conf_level = "LOW"
        print("\n?" + "?" * 78 + "?")
        print(f"? ? YOU SAID:" + " " * 63 + "?")
        print(f"?    \"{text}\"" + " " * (73 - len(text)) + "?")
        print("?" + "?" * 78 + "?")
        print(f"? {conf_emoji} Confidence: {conf_level} ({confidence:.0%})  "
              f"?  ? Language: {language.upper()}  "
              f"?  ??  {duration:.2f}s" + " " * (26 - len(f"{duration:.2f}s")) + "?")
        print("?" + "?" * 78 + "?")
    def simulate_response(self, text: str):
        """Simulate assistant response (placeholder)"""
        text_lower = text.lower()
        responses = {
            'hello': "Hello! How can I help you today?",
            'hi': "Hi there! Nice to hear from you!",
            'how are you': "I'm doing great! Thanks for asking. How are you?",
            'thank': "You're welcome!",
            'bye': "Goodbye! Have a great day!",
            'name': "I'm your companion bot, powered by Whisper voice recognition!",
            'weather': "I don't have weather data yet, but that's a planned feature!",
            'time': f"The current time is {datetime.now().strftime('%I:%M %p')}",
        }
        response = None
        for keyword, reply in responses.items():
            if keyword in text_lower:
                response = reply
                break
        if not response:
            response = "I heard you! (LLM integration coming soon to generate smart responses)"
        print("?" + "?" * 78 + "?")
        print(f"? ? BOT:" + " " * 68 + "?")
        print(f"?    {response}" + " " * (74 - len(response)) + "?")
        print("?" + "?" * 78 + "?")
        print()
    def print_summary(self):
        """Print session summary"""
        stats = self.pipeline.get_statistics()
        print("\n" + "=" * 80)
        print("? SESSION SUMMARY".center(80))
        print("=" * 80)
        print(f"\n? Statistics:")
        print(f"   Total Utterances:         {stats['total_utterances']}")
        print(f"   Total Transcription Time: {stats['total_transcription_time']:.2f}s")
        print(f"   Avg Time per Utterance:   {stats['avg_transcription_time']:.2f}s")
        if self.conversation_history:
            print(f"\n? Conversation History:")
            for i, entry in enumerate(self.conversation_history, 1):
                time_str = entry['timestamp'].strftime("%H:%M:%S")
                print(f"   [{time_str}] #{i}: \"{entry['text']}\" "
                      f"(conf: {entry['confidence']:.0%})")
            avg_conf = sum(e['confidence'] for e in self.conversation_history) / len(self.conversati
on_history)
            print(f"\n   Average Confidence: {avg_conf:.0%}")
        stt_stats = stats.get('stt_stats', {})
        print(f"\n? Whisper Model:")
        print(f"   Model Size: {stt_stats.get('model_size', 'N/A')}")
        print(f"   Device: {stt_stats.get('device', 'N/A')}")
        print("\n" + "=" * 80)
        print("? Demo completed successfully!".center(80))
        print("=" * 80 + "\n")
    def run(self):
        """Run the demo"""
        if not self.start():
            return 1
        try:
            while self.is_running:
                time.sleep(0.1)
        except KeyboardInterrupt:
            pass
        finally:
            self.stop()
        return 0
def main():
    """Main entry point"""
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print("? Config file not found!")
        print(f"Looking for: {config_path}")
        print("\nPlease run setup.sh first or create config/settings.yaml")
        return 1
    try:
        with open(config_path) as f:
            config = yaml.safe_load(f)
    except Exception as e:
        print(f"? Failed to load config: {e}")
        return 1
    demo = VoiceAssistantDemo(config)
    return demo.run()
if __name__ == "__main__":
    sys.exit(main())
```

### tests/test_voice_llm.py

```
#!/usr/bin/env python3
"""
Voice + LLM Test Script
Test voice input with LLM conversation (text output, no TTS)
"""
import sys
import logging
import time
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))
import yaml
from llm.voice_pipeline import VoicePipeline
from llm import ConversationManager
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
def main():
    """Main test function"""
    print("\n" + "="*70)
    print("? VOICE + LLM TEST (Text Output Only)")
    print("="*70)
    config_path = Path(__file__).parent.parent / 'config' / 'settings.yaml'
    if not config_path.exists():
        print("? Config file not found!")
        print(f"   Looking for: {config_path}")
        return 1
    try:
        with open(config_path) as f:
            config = yaml.safe_load(f)
        print("? Configuration loaded")
    except Exception as e:
        print(f"? Failed to load config: {e}")
        return 1
    print("\n? Initializing components...")
    try:
        voice_input = VoicePipeline(config)
        print("   ? Voice input initialized")
        conversation_manager = ConversationManager(config)
        print("   ? Conversation manager initialized")
    except Exception as e:
        print(f"? Failed to initialize components: {e}")
        logger.error("Initialization error", exc_info=True)
        return 1
    if not conversation_manager.llm.is_available:
        print("\n??  WARNING: Ollama not available!")
        print("   Starting Ollama: ollama serve")
        print("   Pull model: ollama pull qwen2.5:0.5b")
        print("\n   Continuing with fallback responses...")
    print("\n" + "="*70)
    print("? LISTENING MODE")
    print("="*70)
    print("\nSpeak into your microphone...")
    print("(The bot will respond with text only, no speech)")
    print("\nPress Ctrl+C to exit")
    print("?" * 70)
    conversation_count = 0
    voice_input.start()
    try:
        while True:
            print("\n? Listening...")
            result = voice_input.wait_for_transcription(timeout=30.0)
            if result is None:
                print("   No speech detected, trying again...")
                continue
            transcription = result.get('text', '').strip()
            confidence = result.get('confidence', 0.0)
            if not transcription:
                print("   Could not transcribe, trying again...")
                continue
            print(f"\n? You said: {transcription} (confidence: {confidence:.0%})")
            if transcription.lower() in ['quit', 'exit', 'goodbye', 'bye']:
                print("\n? Goodbye!")
                break
            print("? Thinking...")
            start_time = time.time()
            response, metadata = conversation_manager.process_user_input(
                transcription
            )
            response_time = time.time() - start_time
            print(f"\n? Bot ({metadata['emotion']}): {response}")
            conversation_count += 1
            if conversation_count % 3 == 0:
                print("\n   ? Metadata:")
                print(f"      ??  Response time: {response_time:.2f}s")
                print(f"      ? Emotion: {metadata['emotion']}")
                print(f"      ? Tokens: {metadata['tokens']}")
                print(f"      ? Energy: {metadata['energy']:.0%}")
                print(f"      ? Messages: {metadata['message_count']}")
                if metadata.get('fallback'):
                    print("      ??  Using fallback response")
            print("\n" + "?" * 70)
    except KeyboardInterrupt:
        print("\n\n??  Stopped by user")
    except Exception as e:
        print(f"\n? Error: {e}")
        logger.error("Runtime error", exc_info=True)
        return 1
    finally:
        voice_input.cleanup()
    print("\n" + "="*70)
    print("SESSION SUMMARY")
    print("="*70)
    summary = conversation_manager.get_conversation_summary()
    print(summary)
    print("\n? Test complete!")
    return 0
if __name__ == "__main__":
    sys.exit(main())
```