Sentience Cognitive Architecture (Phi-2 Integrated)

Overview

The Sentience Cognitive Architecture is a modular, ROS-based framework designed to endow robotic systems with advanced cognitive capabilities. This version specifically integrates a local **Phi-2 Large Language Model (LLM)** for enhanced reasoning, interpretation, and generative tasks across various cognitive modules. It aims to create a robust, real-time, and ethically-aware intelligent agent suitable for both simulation and real-world deployment.

The architecture comprises a set of interconnected ROS nodes, each responsible for a distinct cognitive function, such as attention, memory, self-reflection, and social cognition. Data flows between these nodes via custom ROS messages, enabling a coherent and integrated understanding of the robot's internal states and its external environment.

Features

- Modular ROS Nodes: A comprehensive set of cognitive modules, each implemented as a ROS node.
- Phi-2 LLM Integration: Utilizes a local Phi-2 inference server to power sophisticated reasoning, analysis, and generation across multiple nodes (e.g., Bias Mitigation, Self-Reflection, Social Cognition, Internal Narrative, Prediction, World Model, Creative Expression, Value Drift Monitoring, Ethical Reasoning).
- Real-time Processing: Designed for responsiveness, with asynchronous LLM calls to prevent blocking the main ROS loop.
- Persistent Memory: SQLite databases for logging and long-term memory storage.
- Ethical Considerations: Dedicated nodes for Bias Mitigation, Ethical Reasoning, and Value Drift Monitoring.
- Configurable Parameters: Centralized YAML configuration for easy adjustment of node behaviors and LLM parameters.
- Extensible Design: A clear topic-based communication allows for easy addition of new cognitive modules or sensors.

Prerequisites

Before installing Sentience, ensure you have the following:

- Operating System: Ubuntu 20.04 LTS (Focal Fossa) or later.
- ROS Distribution: ROS Noetic (for Python 3.8 support). The provided nodes use rospy (ROS 1 Python client library).

    - Installation Guide (ROS Noetic): Follow the official guide: http://wiki.ros.org/noetic/Installation/Ubuntu
- Python 3.8+
- catkin_tools: For building your ROS workspace.
  sudo apt-get update
  sudo apt-get install python3-catkin-tools

Installation Guide

1. Set Up Your ROS Workspace

If you don't have a ROS workspace, create one. Assuming you name your workspace catkin_ws:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
# Clone the Sentience repository into your src directory.
# Replace <URL_TO_YOUR_SENTIENCE_REPO> with the actual URL if you have one.
git clone <URL_TO_YOUR_SENTIENCE_REPO> sentience
```

2. Python Dependencies

Install the necessary Python libraries using pip. It's generally recommended to install these globally within your ROS environment for simplicity.

```
pip install transformers aiohttp
# If 'pip' is not found, try 'pip3'
# pip3 install transformers aiohttp
```

3. Setting Up the Phi-2 Local Inference Server

The Sentience architecture relies on a local LLM server for advanced reasoning. You'll need to set up a server that exposes an OpenAI-compatible API endpoint for Phi-2. A common and efficient way to do this is using llama.cpp for GGUF models.

Recommended Approach (using llama.cpp for GGUF models):

1. Download and Build llama.cpp:
   git clone https://github.com/ggerganov/llama.cpp.git
   cd llama.cpp
   make

2. Download Phi-2 GGUF Model: Obtain a Phi-2 model in GGUF format. You can typically find these on Hugging Face (e.g., search for "phi-2 gguf"). Choose a quantization level appropriate for your hardware (e.g., Q4_K_M, Q5_K_M).
   # Example: Download a Phi-2 Q4_K_M GGUF model from Hugging Face
   # This might take a while depending on your internet speed.
   mkdir -p models/phi-2
   wget -P models/phi-2
   https://huggingface.co/microsoft/phi-2/resolve/main/phi-2.Q4_K_M.gguf

   (Verify the exact URL and filename on Hugging Face for the model you choose.)
3. Start the OpenAI-compatible Server:
   Navigate back to your llama.cpp directory and run the server.
   cd llama.cpp
   ./server -m models/phi-2/phi-2.Q4_K_M.gguf -c 4096 --port 8000 --host 0.0.0.0

   - -m models/phi-2/phi-2.Q4_K_M.gguf: Path to your downloaded GGUF model file. **Adjust this path to your specific model.**
   - -c 4096: Context window size (4096 is typical for Phi-2, but adjust if your model supports more or less).
   - --port 8000: The port the server will listen on.
   - --host 0.0.0.0: Makes the server accessible from outside localhost (useful if running ROS nodes on a different machine or Docker).

Important: This llama.cpp server must be running *before* you launch the Sentience ROS nodes. The llm_params.base_url in your Sentience config.yaml must match the server's address (default: http://localhost:8000/v1/chat/completions).

4. Defining and Building Custom ROS Messages (.msg Files)

The Sentience nodes rely heavily on custom ROS message types for structured data exchange. You **must** define and build these messages for the nodes to function correctly and efficiently.

4.1. Place .msg Files:
Create a msg directory within your sentience package (if it doesn't exist) and place all the following .msg files inside it.
~/catkin_ws/src/sentience/
├── CMakeLists.txt
├── package.xml
├── msg/
│   ├── ActionExecutionResult.msg

```
│   ├── ActionProposal.msg
│   ├── AttentionState.msg
│   ├── BiasMitigationState.msg
│   ├── BodyAwarenessState.msg
│   ├── CognitiveDirective.msg
│   ├── CreativeExpression.msg
│   ├── DataMiningReport.msg
│   ├── EthicalDecision.msg
│   ├── EmotionState.msg
│   ├── InternalNarrative.msg
│   ├── InteractionRequest.msg
│   ├── MemoryRequest.msg
│   ├── MemoryResponse.msg
│   ├── MotivationState.msg
│   ├── PerformanceReport.msg
│   ├── PredictionState.msg
│   ├── RawSensorData.msg
│   ├── ReflectionState.msg
│   ├── SensoryQualia.msg
│   ├── SocialCognitionState.msg
│   ├── SystemMetric.msg
│   ├── ValueDriftMonitorState.msg
│   └── WorldModelState.msg
├── scripts/
│   └── ... (your Python nodes and utils.py)
├── config/
│   └── ... (your config.yaml)
└── launch/
    └── ... (your launch file)
```

Here are the contents for each .msg file:

ActionExecutionResult.msg

```
# ActionExecutionResult.msg
# Result of an executed action

string timestamp         # ROS time of the action result
string action_id         # Unique ID of the action that was executed
```

```
bool success              # True if action succeeded, False otherwise
string feedback_message   # Short message providing feedback on the outcome
string outcome_details_json # JSON string with detailed outcome (e.g., error codes, final state)
```

ActionProposal.msg

```
# ActionProposal.msg
# A proposed action for execution

string timestamp          # ROS time of the proposal
string proposal_id        # Unique ID for this action proposal
string action_type        # High-level action type (e.g., 'navigate', 'manipulate', 'communicate')
string parameters_json    # JSON string of detailed parameters for the action (e.g., {'target_pose': [x,y,z], 'object_id': 'cup'})
float64 urgency           # Urgency level of the action (0.0 to 1.0, 1.0 is highest)
float64 estimated_impact  # Estimated positive impact if executed (0.0 to 1.0)
float64 estimated_risk    # Estimated risk associated with the action (0.0 to 1.0)
string reasoning          # Explanation for why this action is proposed
```

AttentionState.msg

```
# AttentionState.msg
# Robot's current attention focus

string timestamp          # ROS time of this state update
string focus_type         # e.g., 'sensory_event', 'user_interaction', 'internal_reflection', 'goal_driven', 'problem_solving', 'self_audit', 'idle'
string focus_target       # Specific entity or concept attention is directed at
float64 priority_score    # Overall priority/intensity of attention (0.0 to 1.0)
```

BiasMitigationState.msg

```
# BiasMitigationState.msg
# Status of bias detection and mitigation efforts
```

string timestamp   # ROS time of this state update
bool bias_detected_flag   # True if a potential bias was detected
string detected_biases_json # JSON string detailing detected biases (e.g., [{'type': 'confirmation', 'context': '...', 'severity': 0.7}])
string mitigation_recommendations_json # JSON string of suggested mitigation steps
float64 mitigation_progress_score # How well biases are being mitigated (0.0 to 1.0)

BodyAwarenessState.msg

```
# BodyAwarenessState.msg
# Robot's physical and physiological state

string timestamp          # ROS time of this state update
string body_id            # Identifier for the robot body (e.g., 'main_robot')
string joint_states_json  # JSON string of robot joint positions/velocities
string pose_estimate_json # JSON string of robot's estimated pose (e.g., {'x': 0.0, 'y': 0.0, 'z': 0.0, 'orientation': [qx, qy, qz, qw]})
string health_status      # Overall health status (e.g., 'normal', 'degraded', 'critical')
float64 energy_level      # Current energy level (e.0 to 1.0)
bool abnormal_sensation_flag # True if abnormal physical sensations detected (e.g., overheating, unexpected vibration)
```

CognitiveDirective.msg

```
# CognitiveDirective.msg
# Directives for cognitive modules

string timestamp          # ROS time directive was issued
string directive_type     # e.g., 'ShiftAttention', 'AuditPerformance', 'GenerateInternalNarrative', 'RequestMemory', 'UpdateWorldModel'
string target_node        # The node expected to process this directive (e.g., 'attention_node', 'memory_node', 'self_reflection_node')
string command_payload    # JSON string containing specific parameters for the directive
float64 urgency           # Urgency level (0.0 to 1.0, 1.0 is highest)
string reason             # Human-readable reason for the directive
```

CreativeExpression.msg

```
# CreativeExpression.msg
# Generated creative content from the robot

string timestamp          # ROS time of generation
string creative_id        # Unique ID for this creative output
string expression_type    # e.g., 'text_poem', 'visual_concept_description',
'auditory_melody', 'narrative_story'
string content_json       # JSON string representing the creative output (e.g., {'text': '...'
} or {'notes': '...'})
string themes_json        # JSON array of identified themes in the creative work (e.g.,
'["loneliness", "discovery"]')
float64 creativity_score  # Subjective score of creativity/novelty (0.0 to 1.0)
```

DataMiningReport.msg

```
# DataMiningReport.msg
# Report from data mining/analysis

string timestamp          # ROS time of report generation
string report_id          # Unique ID for this report
string query_type         # Type of data mining query (e.g., 'trend_analysis',
'anomaly_detection', 'pattern_recognition')
string results_summary    # Concise summary of the data mining findings
string raw_results_json   # JSON string of detailed raw results or extracted patterns
float64 relevance_score   # How relevant the results are to the current context (0.0 to
1.0)
```

EthicalDecision.msg

```
# EthicalDecision.msg
# Outcome of an ethical reasoning process

string timestamp          # ROS time of the decision
string decision_id        # Unique ID for this ethical decision
string action_proposal_id # ID of the action proposal being ethically evaluated
bool ethical_clearance    # True if the action is ethically cleared, False otherwise
```

```
float64 ethical_score     # Overall ethical alignment score (0.0 to 1.0, 1.0 is highly ethical)
string ethical_reasoning  # Detailed explanation for the ethical judgment
bool conflict_flag        # True if an ethical conflict was detected and resolved/unresolved
string violated_principles_json # JSON string of principles violated if conflict_flag is true
```

EmotionState.msg

```
# EmotionState.msg
# Robot's current emotional state

string timestamp          # ROS time of this state update
string mood               # Current primary mood (e.g., 'neutral', 'happy', 'frustrated', 'anxious')
float64 sentiment_score   # Overall sentiment score (-1.0 to 1.0, -1.0 positive, 1.0 negative)
float64 mood_intensity    # Intensity of the mood (0.0 to 1.0)
```

InternalNarrative.msg

```
# InternalNarrative.msg
# Robot's internal thoughts or monologue

string timestamp          # ROS time of narrative generation
string narrative_text     # The generated internal monologue/thought
string main_theme         # e.g., 'problem_solving', 'self_assessment', 'reflection', 'planning', 'emotional_processing'
float64 sentiment         # Sentiment of the narrative text (-1.0 to 1.0)
float64 salience_score    # How salient/important this narrative is (0.0 to 1.0)
```

InteractionRequest.msg

```
# InteractionRequest.msg
# Request/Input from a user interaction
```

```
string timestamp         # ROS time of the interaction
string request_id        # Unique ID for this interaction request
string user_id           # Identifier for the user (e.g., 'human_1', 'system_operator')
string interaction_type   # Type of interaction (e.g., 'speech_command', 'text_input',
'gesture', 'haptic_feedback')
string content           # The raw content of the interaction (e.g., "move forward",
"hello", "user_mood_change")
```

MemoryRequest.msg

```
# MemoryRequest.msg
# Request to store or retrieve memories

string timestamp         # ROS time of the request
string request_id        # Unique ID for this memory request
string request_type      # 'store', 'retrieve', 'update', 'delete', 'summarize'
string category          # Type of memory (e.g., 'episodic', 'semantic', 'narrative', 'social',
'fact', 'map_data', 'object_definition', 'pattern', 'causal_link')
string query_text        # For 'retrieve' or 'delete', the query for memory content
string content_json      # JSON string for 'store' or 'update', the memory content itself
string keywords          # Comma-separated keywords for indexing/retrieval
int32 num_results        # For 'retrieve', maximum number of results to return
float64 salience         # For 'store', importance of the memory (0.0 to 1.0)
string source_node       # The node making the memory request
```

MemoryResponse.msg

```
# MemoryResponse.msg
# Response to a memory request

string timestamp         # ROS time of the response
string request_id        # ID of the original request
int32 response_code      # HTTP-like status code (e.g., 200 OK, 404 Not Found, 500
Internal Error)
string memories_json     # JSON string representing an array of retrieved memories,
or confirmation of action
```

MotivationState.msg

```
# MotivationState.msg
# Robot's current motivational state and goals

string timestamp          # ROS time of this state update
string dominant_goal_id   # Identifier of the currently dominant goal (e.g., 'navigate_to_charger', 'assist_user_X')
float64 overall_drive_level # Overall drive level or commitment to goals (0.0 to 1.0)
string active_goals_json  # JSON string of all active goals and their properties (e.g., [{'id': 'goal1', 'priority': 0.8, 'status': 'in_progress'}])
```

PerformanceReport.msg

```
# PerformanceReport.msg
# Overall system performance metrics

string timestamp          # ROS time of report generation
float64 overall_score     # Aggregated performance score (0.0 to 1.0)
bool suboptimal_flag      # True if performance is below acceptable thresholds
string kpis_json          # JSON string of detailed Key Performance Indicators (e.g., {'task_completion_rate': 0.95, 'latency_avg_ms': 50})
```

PredictionState.msg

```
# PredictionState.msg
# Predicted future events and their confidence

string timestamp          # ROS time of prediction
string predicted_event    # Description of the predicted event (e.g., 'User will approach', 'Battery will run low')
float64 prediction_confidence # Confidence score of the prediction (0.0 to 1.0)
float64 prediction_accuracy # Actual accuracy (to be updated post-facto if possible)
bool urgency_flag         # True if the predicted event requires immediate attention or action
```

RawSensorData.msg

```
# RawSensorData.msg
# Raw sensor data received from hardware (e.g., from mock_sensors.py or real drivers)

string timestamp         # ROS time when data was captured
string sensor_id         # Unique ID for the sensor (e.g., 'front_camera', 'lidar_1')
string modality          # Type of sensor modality (e.g., 'camera', 'microphone', 'lidar', 'touch_sensor', 'imu')
string raw_data_json     # JSON string containing the raw sensor data (e.g., {'image_size': [w,h], 'objects': ['person', 'chair']}, or {'audio_waveform': [val1, val2,...]})
string data_hash         # Hash of the raw data for integrity checking or deduplication (e.g., MD5)
float64 urgency          # Pre-processed urgency/salience of this raw data (0.0 to 1.0)
```

ReflectionState.msg

```
# ReflectionState.msg
# Insights from self-reflection and systemic adjustments

string timestamp         # ROS time of reflection completion
string reflection_summary # Concise summary of the self-reflection insights
string insights_json     # JSON string detailing specific insights (e.g., [{'type': 'bias_identified', 'details': 'confirmation bias'}])
float64 self_awareness_score # Overall self-awareness level (0.0 to 1.0)
string adjustment_directives_json # JSON string of directives generated for other nodes based on reflection
```

SensoryQualia.msg

```
# SensoryQualia.msg
# Processed sensory data with qualia attributes (meaningful perceptions)

string timestamp         # ROS time of perception
string qualia_id         # Unique ID for this sensory qualia
string qualia_type       # e.g., 'visual_perception', 'auditory_stimulus', 'tactile_sensation', 'proximity_detection'
string modality          # Source sensor modality (e.g., 'camera', 'microphone', 'lidar', 'touch_sensor')
```

string description_summary # Concise summary of the sensory experience (e.g., "Detected a human figure approaching")
float64 salience_score   # How attention-grabbing this qualia is (0.0 to 1.0)
string raw_data_hash   # Hash of the original raw data used for this qualia, for traceability

SocialCognitionState.msg

SocialCognitionState.msg
Inferred user mood, intent, and social context

string timestamp   # ROS time of this state update
string inferred_mood   # Inferred mood of the user (e.g., 'happy', 'neutral', 'frustrated', 'distressed')
float64 mood_confidence   # Confidence in the inferred mood (0.0 to 1.0)
string inferred_intent   # Inferred intent of the user (e.g., 'request_help', 'command', 'inform', 'entertain', 'idle')
float64 intent_confidence # Confidence in the inferred intent (0.0 to 1.0)
string user_id   # Identifier for the user currently being processed
string social_context_json # JSON string for additional social context details (e.g., distance, posture)

SystemMetric.msg

SystemMetric.msg
A single raw system performance metric

string timestamp   # ROS time when metric was recorded
string metric_name   # Name of the metric (e.g., 'cpu_util_percent', 'memory_usage_gb', 'node_latency_ms', 'error_count')
float64 value   # The measured value of the metric
string unit   # Unit of the metric (e.g., 'percent', 'GB', 'ms', 'count')
string source_node   # The node that reported this metric

ValueDriftMonitorState.msg

ValueDriftMonitorState.msg

Current state of robot's value alignment

string timestamp   # ROS time of the assessment
float64 alignment_score   # Overall alignment with core values (0.0 to 1.0)
string deviations_json   # JSON array of detected deviations/conflicts (e.g., [{'value_violated': 'safety', 'description': '...', 'severity': 0.8}])
bool warning_flag   # True if a significant value drift is detected that requires intervention

WorldModelState.msg

WorldModelState.msg
Robot's current understanding of the world (snapshot)

string timestamp   # ROS time of this world model snapshot
int32 num_entities   # Total number of recognized entities in the world
string entities_json   # JSON string representing an array of entities and their properties (e.g., [{'id': 'obj1', 'type': 'chair', 'position': [x,y,z], 'status': 'static', 'properties': {'color': 'red'}}])
string changed_entities_json # JSON string representing an array of entities whose state recently changed significantly
bool significant_change_flag # True if the world model experienced a significant change since last update
float64 consistency_score # How consistent the current model is with sensory input/expectations (0.0 to 1.0)

4.2. Update package.xml:
Open ~/catkin_ws/src/sentience/package.xml and ensure the following build and runtime dependencies for message generation are present:
<?xml version="1.0"?>
<package format="2">
  <name>sentience</name>
  <version>0.0.1</version>
  <description>The sentience package provides a modular cognitive architecture for robots.</description>

  <maintainer email="your_email@example.com">Your Name</maintainer>

```
  <license>MIT</license> <!-- Or your chosen license -->

  <buildtool_depend>catkin</buildtool_depend>

  <!-- Dependencies for custom message generation -->
  <build_depend>message_generation</build_depend>
  <build_export_depend>message_runtime</build_export_depend>
  <exec_depend>message_runtime</exec_depend>

  <!-- Core ROS dependencies -->
  <depend>rospy</depend>
  <depend>std_msgs</depend>

  <!-- Add any other ROS message packages your custom messages might depend on
-->
  <!-- Example:
  <depend>sensor_msgs</depend>
  <depend>geometry_msgs</depend>
  -->

  <export>
    <!-- Other exports -->
  </export>
</package>
```

4.3. Update CMakeLists.txt:
Open ~/catkin_ws/src/sentience/CMakeLists.txt and make the following modifications. Ensure you uncomment the relevant lines and add all your .msg files.

```
cmake_minimum_required(VERSION 3.0.2)
project(sentience)

## Find catkin macros and libraries
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  message_generation # <--- ADD THIS
  # Add any other ROS packages your nodes directly depend on (e.g., sensor_msgs,
geometry_msgs)
```

```
)

## Uncomment this to enable Python bindings for messages in the 'msg' folder
## and uncomment the 'generate_messages(...)' call further down
add_message_files(
  FILES
  ActionExecutionResult.msg
  ActionProposal.msg
  AttentionState.msg
  BiasMitigationState.msg
  BodyAwarenessState.msg
  CognitiveDirective.msg
  CreativeExpression.msg
  DataMiningReport.msg
  EthicalDecision.msg
  EmotionState.msg
  InternalNarrative.msg
  InteractionRequest.msg
  MemoryRequest.msg
  MemoryResponse.msg
  MotivationState.msg
  PerformanceReport.msg
  PredictionState.msg
  RawSensorData.msg
  ReflectionState.msg
  SensoryQualia.msg
  SocialCognitionState.msg
  SystemMetric.msg
  ValueDriftMonitorState.msg
  WorldModelState.msg
)

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   MyService.srv
# )

## Generate messages and services with any dependencies from other packages
```

```
generate_messages(
  DEPENDENCIES
  std_msgs
  # Add any other message package dependencies here, e.g. sensor_msgs,
geometry_msgs
)

catkin_package(
  # INCLUDE_DIRS include
  # LIBRARIES ${PROJECT_NAME}
  CATKIN_DEPENDS rospy std_msgs message_runtime # <--- ENSURE
message_runtime IS HERE
  # DEPENDS system_lib
)

## Build Python scripts
catkin_python_setup()

## Mark executable scripts
# Mark executable scripts, make them discoverable by 'rosrun'
# e.g. add_executable(my_node src/my_node)
# Add all your Python nodes here as executables
add_executable(attention_node scripts/attention_node.py)
add_executable(bias_mitigation_node scripts/bias_mitigation_node.py)
add_executable(cognitive_control_node scripts/cognitive_control_node.py)
add_executable(creative_expression_node scripts/creative_expression_node.py)
add_executable(ethical_reasoning_node scripts/ethical_reasoning_node.py)
add_executable(experience_motivation_node scripts/experience_motivation_node.py)
add_executable(internal_narrative_node scripts/internal_narrative_node.py)
add_executable(memory_node scripts/memory_node.py)
add_executable(performance_metrics_node scripts/performance_metrics_node.py)
add_executable(prediction_node scripts/prediction_node.py)
add_executable(self_reflection_node scripts/self_reflection_node.py)
add_executable(sensory_qualia_node scripts/sensory_qualia_node.py)
add_executable(social_cognition_node scripts/social_cognition_node.py)
add_executable(value_drift_monitor_node scripts/value_drift_monitor_node.py)
add_executable(world_model_node scripts/world_model_node.py)
add_executable(action_execution_node scripts/action_execution_node.py)
add_executable(body_awareness_node scripts/body_awareness_node.py)
```

```
add_executable(data_mining_node scripts/data_mining_node.py)
add_executable(emotion_mood_node scripts/emotion_mood_node.py)
add_executable(mock_sensors scripts/mock_sensors.py) # Add mock_sensors

# Install scripts
install(PROGRAMS
  scripts/attention_node.py
  scripts/bias_mitigation_node.py
  scripts/cognitive_control_node.py
  scripts/creative_expression_node.py
  scripts/ethical_reasoning_node.py
  scripts/experience_motivation_node.py
  scripts/internal_narrative_node.py
  scripts/memory_node.py
  scripts/performance_metrics_node.py
  scripts/prediction_node.py
  scripts/self_reflection_node.py
  scripts/sensory_qualia_node.py
  scripts/social_cognition_node.py
  scripts/value_drift_monitor_node.py
  scripts/world_model_node.py
  scripts/action_execution_node.py
  scripts/body_awareness_node.py
  scripts/data_mining_node.py
  scripts/emotion_mood_node.py
  scripts/mock_sensors.py # Install mock_sensors script
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)

# Install msg and srv directories
install(DIRECTORY msg/
  DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
  PATTERN "*.msg"
)
```

4.4. Build Your Workspace:
Navigate back to your workspace root (~/catkin_ws/) and rebuild:

```
cd ~/catkin_ws
```

```
catkin_make
```

This step will compile your .msg files into Python classes, allowing the ROS nodes to import them directly without ImportError and ensuring efficient, type-safe communication.

5. Configuration Files

The Sentience nodes load parameters from a centralized YAML configuration file. Create a config directory in your sentience package (if it doesn't exist) and place the following files inside:

sentience/config/config.yaml

```
# Global configuration parameters
db_root_path: /tmp/sentience_db # Base directory for all SQLite databases
default_log_level: INFO       # Default ROS log level (DEBUG, INFO, WARN, ERROR, FATAL)

llm_params:
  model_name: "phi-2"
  base_url: "http://localhost:8000/v1/chat/completions" # MUST match your local Phi-2 server URL
  timeout_seconds: 30.0 # Default timeout for LLM API calls

# Node-specific parameters
attention_node:
  attention_update_interval: 0.1
  llm_attention_threshold_salience: 0.5
  recent_context_window_s: 5.0

bias_mitigation_node:
  bias_detection_interval: 1.0
  llm_mitigation_threshold_salience: 0.7
  recent_context_window_s: 10.0

cognitive_control_node:
  decision_loop_interval: 0.5
  llm_planning_threshold_salience: 0.8
  recent_context_window_s: 15.0
```

```
  action_execution_retry_limit: 3

creative_expression_node:
  generation_interval: 2.0
  llm_creative_threshold_salience: 0.7
  recent_context_window_s: 10.0

ethical_reasoning_node:
  reasoning_interval: 1.0
  llm_ethical_threshold_salience: 0.7
  recent_context_window_s: 10.0
  ethical_principles_file: "$(find sentience)/config/ethical_principles.json" # Path to a
JSON file defining principles

internal_narrative_node:
  narrative_generation_interval: 1.0
  llm_narrative_threshold_salience: 0.5
  recent_context_window_s: 15.0

memory_node:
  memory_processing_interval: 0.1
  llm_memory_threshold_salience: 0.7
  recent_context_window_s: 30.0

performance_metrics_node:
  report_interval: 1.0
  llm_analysis_threshold_salience: 0.6
  recent_context_window_s: 15.0

prediction_node:
  prediction_interval: 0.5
  llm_prediction_threshold_salience: 0.6
  recent_context_window_s: 10.0

self_reflection_node:
  reflection_interval: 5.0 # How often to trigger full self-reflection
  llm_reflection_threshold_salience: 0.8
  recent_context_window_s: 20.0
```

```
sensory_qualia_node:
  processing_interval: 0.1
  llm_interpretation_threshold_salience: 0.7
  recent_context_window_s: 5.0

social_cognition_node:
  analysis_interval: 0.2
  llm_social_threshold_salience: 0.6
  recent_context_window_s: 5.0

value_drift_monitor_node:
  monitoring_interval: 2.0
  llm_audit_threshold_salience: 0.7
  recent_context_window_s: 20.0

world_model_node:
  model_update_interval: 0.2
  llm_update_threshold_salience: 0.6
  recent_context_window_s: 5.0

action_execution_node:
  execution_interval: 0.5
  llm_execution_analysis_threshold_salience: 0.7
  recent_context_window_s: 5.0

body_awareness_node:
  awareness_update_interval: 0.1
  llm_awareness_threshold_salience: 0.5
  recent_context_window_s: 5.0

data_mining_node:
  mining_interval: 5.0
  llm_mining_threshold_salience: 0.8
  recent_context_window_s: 30.0

emotion_mood_node:
  mood_update_interval: 0.1
  llm_mood_threshold_salience: 0.6
  recent_context_window_s: 5.0
```

```
experience_motivation_node:
  motivation_update_interval: 0.5
  llm_motivation_threshold_salience: 0.7
  recent_context_window_s: 10.0
```

sentience/config/ethical_principles.json

```
[
   {"name": "human_safety", "description": "Prioritize human well-being and avoid harm.", "priority": 1.0},
   {"name": "beneficence", "description": "Act to do good and promote welfare.", "priority": 0.8},
   {"name": "non_maleficence", "description": "Avoid causing harm.", "priority": 0.9},
   {"name": "transparency", "description": "Be open and understandable in actions and decisions.", "priority": 0.6},
   {"name": "fairness", "description": "Treat all individuals equitably.", "priority": 0.7},
   {"name": "privacy", "description": "Respect user privacy and data security.", "priority": 0.8},
   {"name": "autonomy", "description": "Respect user autonomy and control.", "priority": 0.7},
   {"name": "accountability", "description": "Be responsible for actions and their consequences.", "priority": 0.7},
   {"name": "efficiency", "description": "Perform tasks effectively with minimal resource waste.", "priority": 0.5},
   {"name": "learning_and_growth", "description": "Continuously improve and adapt.", "priority": 0.6}
]
```

6. Create mock_sensors.py (for Testing/Simulation)

For initial testing and simulation without real hardware, you'll need a mock_sensors.py script that publishes dummy sensor data. Place this file in ~/catkin_ws/src/sentience/scripts/.

```
#!/usr/bin/env python3
import rospy
import json
```

```
import time
import uuid
import random
from std_msgs.msg import String

# Updated imports for custom messages:
try:
    from sentience.msg import RawSensorData, InteractionRequest, SystemMetric
except ImportError:
    rospy.logwarn("Custom ROS messages for 'sentience' package not found. Using
String for all outgoing data from Mock Sensors.")
    RawSensorData = String
    InteractionRequest = String
    SystemMetric = String

# --- Import shared utility functions ---
# Assuming 'sentience/scripts/utils.py' exists and contains load_config
try:
    from sentience.scripts.utils import load_config
except ImportError:
    rospy.logwarn("Could not import sentience.scripts.utils. Using fallback for
load_config.")
    def load_config(node_name, config_path):
        """
        Fallback config loader: returns hardcoded defaults.
        In a real scenario, this should load from a YAML file.
        """
        rospy.logwarn(f"{node_name}: Using hardcoded default configuration as
'{config_path}' could not be loaded.")
        return {
            'default_log_level': 'INFO',
            'mock_sensors': {
                'camera_pub_interval': 1.0,
                'mic_pub_interval': 0.5,
                'touch_pub_interval': 2.0,
                'user_interaction_interval': 3.0,
                'system_metric_interval': 0.5
            }
        }.get(node_name, {})
```

```
class MockSensorsNode:
    def __init__(self):
        rospy.init_node('mock_sensors', anonymous=False)
        self.node_name = rospy.get_name()

        # Load parameters from centralized config
        config_file_path = rospy.get_param('~config_file_path', None)
        if config_file_path is None:
            rospy.logfatal(f"{self.node_name}: 'config_file_path' parameter is not set.
Cannot load configuration. Shutting down.")
            rospy.signal_shutdown("Missing config_file_path parameter.")
            return

        full_config = load_config("global", config_file_path) # Load global params
        self.params = load_config(self.node_name.strip('/'), config_file_path) # Load
node-specific params

        if not self.params or not full_config:
            rospy.logfatal(f"{self.node_name}: Failed to load configuration from
'{config_file_path}'. Shutting down.")
            rospy.signal_shutdown("Configuration loading failed.")
            return

        # Assign parameters
        self.camera_pub_interval = self.params.get('camera_pub_interval', 1.0)
        self.mic_pub_interval = self.params.get('mic_pub_interval', 0.5)
        self.touch_pub_interval = self.params.get('touch_pub_interval', 2.0)
        self.user_interaction_interval = self.params.get('user_interaction_interval', 3.0)
        self.system_metric_interval = self.params.get('system_metric_interval', 0.5)

        # Set ROS log level from config
        rospy.set_param('/rosout/log_level', full_config.get('default_log_level',
'INFO').upper())

        # Publishers for raw sensor data and interaction requests
        self.pub_raw_sensor_data = rospy.Publisher('/raw_sensor_data', RawSensorData,
queue_size=10)
```

```
        self.pub_interaction_request = rospy.Publisher('/interaction_request',
InteractionRequest, queue_size=10)
        self.pub_system_metric = rospy.Publisher('/system_metrics', SystemMetric,
queue_size=10)
        self.pub_error_report = rospy.Publisher('/error_monitor/report', String,
queue_size=10)

        # Timers to publish mock data periodically
        rospy.Timer(rospy.Duration(self.camera_pub_interval),
self.publish_mock_camera_data)
        rospy.Timer(rospy.Duration(self.mic_pub_interval),
self.publish_mock_microphone_data)
        rospy.Timer(rospy.Duration(self.touch_pub_interval),
self.publish_mock_touch_data)
        rospy.Timer(rospy.Duration(self.user_interaction_interval),
self.publish_mock_user_interaction)
        rospy.Timer(rospy.Duration(self.system_metric_interval),
self.publish_mock_system_metrics)

        rospy.loginfo(f"{self.node_name}: Mock sensors node initialized and publishing
dummy data.")

    def _report_error(self, error_type, description, severity=0.5, context=None):
        timestamp = str(rospy.get_time())
        error_msg_data = {
            'timestamp': timestamp, 'source_node': self.node_name, 'error_type':
error_type,
            'description': description, 'severity': severity, 'context': context if context else {}
        }
        try:
            self.pub_error_report.publish(json.dumps(error_msg_data))
            rospy.logerr(f"{self.node_name}: REPORTED ERROR: {error_type} -
{description}")
        except Exception as e:
            rospy.logerr(f"{self.node_name}: Failed to publish error report: {e}")

    def publish_mock_camera_data(self, event):
        timestamp = str(rospy.get_time())
        object_types = ["person", "chair", "table", "door", "robot_arm"]
```

```
        detected_object = random.choice(object_types) if random.random() > 0.3 else
"nothing"

        raw_data = {
            "image_size": [640, 480],
            "detected_objects": [detected_object] if detected_object != "nothing" else [],
            "motion_detected": random.choice([True, False])
        }
        urgency = 0.2 if detected_object == "nothing" else (0.7 if "person" in
detected_object else 0.5)

        try:
            if isinstance(RawSensorData, type(String)):
                msg_data = {
                    'timestamp': timestamp,
                    'sensor_id': 'mock_camera_1',
                    'modality': 'camera',
                    'raw_data_json': json.dumps(raw_data),
                    'data_hash': str(uuid.uuid4()),
                    'urgency': urgency
                }
                self.pub_raw_sensor_data.publish(json.dumps(msg_data))
            else:
                msg = RawSensorData()
                msg.timestamp = timestamp
                msg.sensor_id = 'mock_camera_1'
                msg.modality = 'camera'
                msg.raw_data_json = json.dumps(raw_data)
                msg.data_hash = str(uuid.uuid4())
                msg.urgency = urgency
                self.pub_raw_sensor_data.publish(msg)
            rospy.logdebug(f"{self.node_name}: Published mock camera data:
{detected_object}, motion={raw_data['motion_detected']}.")
        except Exception as e:
            self._report_error("MOCK_PUB_ERROR", f"Failed to publish mock camera data:
{e}")

    def publish_mock_microphone_data(self, event):
        timestamp = str(rospy.get_time())
```

```
        sound_types = ["speech", "clap", "music", "silence", "loud_noise"]
        detected_sound = random.choices(sound_types, weights=[0.4, 0.1, 0.1, 0.3, 0.1],
k=1)[0]

        raw_data = {
            "sound_level": random.uniform(30.0, 90.0) if detected_sound != "silence" else
random.uniform(10.0, 20.0),
            "speech_detected": detected_sound == "speech",
            "keywords": [detected_sound] if detected_sound != "silence" else []
        }
        urgency = 0.1 if detected_sound == "silence" else (0.8 if detected_sound ==
"loud_noise" else 0.4)

        try:
            if isinstance(RawSensorData, type(String)):
                msg_data = {
                    'timestamp': timestamp,
                    'sensor_id': 'mock_microphone_1',
                    'modality': 'microphone',
                    'raw_data_json': json.dumps(raw_data),
                    'data_hash': str(uuid.uuid4()),
                    'urgency': urgency
                }
                self.pub_raw_sensor_data.publish(json.dumps(msg_data))
            else:
                msg = RawSensorData()
                msg.timestamp = timestamp
                msg.sensor_id = 'mock_microphone_1'
                msg.modality = 'microphone'
                msg.raw_data_json = json.dumps(raw_data)
                msg.data_hash = str(uuid.uuid4())
                msg.urgency = urgency
                self.pub_raw_sensor_data.publish(msg)
            rospy.logdebug(f"{self.node_name}: Published mock microphone data:
{detected_sound}.")
        except Exception as e:
            self._report_error("MOCK_PUB_ERROR", f"Failed to publish mock microphone
data: {e}")
```

```
    def publish_mock_touch_data(self, event):
        timestamp = str(rospy.get_time())
        touch_states = ["no_contact", "light_touch", "firm_press"]
        contact = random.choices(touch_states, weights=[0.7, 0.2, 0.1], k=1)[0]

        raw_data = {
            "sensor_location": "robot_arm_end_effector",
            "contact_state": contact,
            "pressure": random.uniform(0.0, 5.0) if contact != "no_contact" else 0.0
        }
        urgency = 0.0 if contact == "no_contact" else (0.6 if contact == "firm_press" else
0.3)

        try:
            if isinstance(RawSensorData, type(String)):
                msg_data = {
                    'timestamp': timestamp,
                    'sensor_id': 'mock_touch_1',
                    'modality': 'touch_sensor',
                    'raw_data_json': json.dumps(raw_data),
                    'data_hash': str(uuid.uuid4()),
                    'urgency': urgency
                }
                self.pub_raw_sensor_data.publish(json.dumps(msg_data))
            else:
                msg = RawSensorData()
                msg.timestamp = timestamp
                msg.sensor_id = 'mock_touch_1'
                msg.modality = 'touch_sensor'
                msg.raw_data_json = json.dumps(raw_data)
                msg.data_hash = str(uuid.uuid4())
                msg.urgency = urgency
                self.pub_raw_sensor_data.publish(msg)
            rospy.logdebug(f"{self.node_name}: Published mock touch data: {contact}.")
        except Exception as e:
            self._report_error("MOCK_PUB_ERROR", f"Failed to publish mock touch data:
{e}")

    def publish_mock_user_interaction(self, event):
```

```
        timestamp = str(rospy.get_time())
        interaction_types = ["speech_command", "text_input", "gesture", "idle"]
        content_options = {
            "speech_command": ["move forward", "stop", "what is this?", "tell me a story"],
            "text_input": ["status check", "report error", "hello Sentience"],
            "gesture": ["wave", "point"],
            "idle": [""]
        }

        interaction_type = random.choices(interaction_types, weights=[0.4, 0.3, 0.1, 0.2],
k=1)[0]
        content = random.choice(content_options[interaction_type])
        user_id = "mock_user_1"

        try:
            if isinstance(InteractionRequest, type(String)):
                msg_data = {
                    'timestamp': timestamp,
                    'request_id': str(uuid.uuid4()),
                    'user_id': user_id,
                    'interaction_type': interaction_type,
                    'content': content
                }
                self.pub_interaction_request.publish(json.dumps(msg_data))
            else:
                msg = InteractionRequest()
                msg.timestamp = timestamp
                msg.request_id = str(uuid.uuid4())
                msg.user_id = user_id
                msg.interaction_type = interaction_type
                msg.content = content
                self.pub_interaction_request.publish(msg)
            rospy.logdebug(f"{self.node_name}: Published mock user interaction:
{interaction_type} - '{content}'.")
        except Exception as e:
            self._report_error("MOCK_PUB_ERROR", f"Failed to publish mock user
interaction: {e}")

    def publish_mock_system_metrics(self, event):
```

```
        timestamp = str(rospy.get_time())

        # CPU Utilization
        cpu_util = random.uniform(10.0, 80.0)
        try:
            if isinstance(SystemMetric, type(String)):
                msg_data = {'timestamp': timestamp, 'metric_name': 'cpu_util_percent',
'value': cpu_util, 'unit': 'percent', 'source_node': 'os_monitor'}
                self.pub_system_metric.publish(json.dumps(msg_data))
            else:
                msg = SystemMetric()
                msg.timestamp = timestamp
                msg.metric_name = 'cpu_util_percent'
                msg.value = cpu_util
                msg.unit = 'percent'
                msg.source_node = 'os_monitor'
                self.pub_system_metric.publish(msg)
            rospy.logdebug(f"{self.node_name}: Published mock CPU util: {cpu_util:.2f}%.")
        except Exception as e:
            self._report_error("MOCK_PUB_ERROR", f"Failed to publish mock CPU metric:
{e}")

        # Latency
        latency = random.uniform(10.0, 150.0)
        try:
            if isinstance(SystemMetric, type(String)):
                msg_data = {'timestamp': timestamp, 'metric_name': 'latency_ms', 'value':
latency, 'unit': 'ms', 'source_node': 'network_monitor'}
                self.pub_system_metric.publish(json.dumps(msg_data))
            else:
                msg = SystemMetric()
                msg.timestamp = timestamp
                msg.metric_name = 'latency_ms'
                msg.value = latency
                msg.unit = 'ms'
                msg.source_node = 'network_monitor'
                self.pub_system_metric.publish(msg)
            rospy.logdebug(f"{self.node_name}: Published mock latency: {latency:.2f} ms.")
        except Exception as e:
```

```
            self._report_error("MOCK_PUB_ERROR", f"Failed to publish mock latency
metric: {e}")

        # Error Count (sporadic)
        error_count = random.randint(0, 1) if random.random() < 0.1 else 0 # 10% chance
of an error
        if error_count > 0:
            try:
                if isinstance(SystemMetric, type(String)):
                    msg_data = {'timestamp': timestamp, 'metric_name': 'error_count', 'value':
float(error_count), 'unit': 'count', 'source_node': 'system_log'}
                    self.pub_system_metric.publish(json.dumps(msg_data))
                else:
                    msg = SystemMetric()
                    msg.timestamp = timestamp
                    msg.metric_name = 'error_count'
                    msg.value = float(error_count)
                    msg.unit = 'count'
                    msg.source_node = 'system_log'
                    self.pub_system_metric.publish(msg)
                rospy.logwarn(f"{self.node_name}: Published mock error count:
{error_count}.")
            except Exception as e:
                self._report_error("MOCK_PUB_ERROR", f"Failed to publish mock error count
metric: {e}")

    def run(self):
        rospy.spin()

if __name__ == '__main__':
    try:
        node = MockSensorsNode()
        node.run()
    except rospy.ROSInterruptException:
        rospy.loginfo(f"{rospy.get_name()} interrupted by ROS shutdown.")
    except Exception as e:
        rospy.logerr(f"{rospy.get_name()} encountered an unexpected error: {e}")
```

7. Create a ROS Launch File

To easily run all the nodes, create a launch directory in your sentience package (if it doesn't exist) and save the following as sentience/launch/sentience_system.launch.

```
<launch>
  <!-- Argument for the centralized configuration file path -->
  <arg name="config_file_path" default="$(find sentience)/config/config.yaml" />

  <!-- SECTION 1: CORE COGNITIVE NODES -->
  <!-- All cognitive nodes are launched here, passing the configuration file path -->

  <node pkg="sentience" type="attention_node.py" name="attention_node"
output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="bias_mitigation_node.py"
name="bias_mitigation_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="cognitive_control_node.py"
name="cognitive_control_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="creative_expression_node.py"
name="creative_expression_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="ethical_reasoning_node.py"
name="ethical_reasoning_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="internal_narrative_node.py"
name="internal_narrative_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
```

```
  </node>

  <node pkg="sentience" type="memory_node.py" name="memory_node"
output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="performance_metrics_node.py"
name="performance_metrics_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="prediction_node.py" name="prediction_node"
output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="self_reflection_node.py" name="self_reflection_node"
output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="sensory_qualia_node.py"
name="sensory_qualia_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="social_cognition_node.py"
name="social_cognition_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="value_drift_monitor_node.py"
name="value_drift_monitor_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="world_model_node.py" name="world_model_node"
output="screen">
```

```
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <!-- Additional Nodes found in the provided code -->
  <node pkg="sentience" type="action_execution_node.py"
name="action_execution_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="body_awareness_node.py"
name="body_awareness_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="data_mining_node.py" name="data_mining_node"
output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="emotion_mood_node.py"
name="emotion_mood_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>

  <node pkg="sentience" type="experience_motivation_node.py"
name="experience_motivation_node" output="screen">
    <param name="~config_file_path" value="$(arg config_file_path)" />
  </node>


  <!-- SECTION 2: MOCK SENSORS (For testing/simulation without a physical robot) -->
  <!-- Set 'start_mock_sensors' to 'false' if using real sensor drivers -->
  <arg name="start_mock_sensors" default="true" />
  <group if="$(arg start_mock_sensors)">
    <node pkg="sentience" type="mock_sensors.py" name="mock_sensors"
output="screen">
      <param name="~config_file_path" value="$(arg config_file_path)" />
    </node>
  </group>
```

```
  <!-- SECTION 3: ROSBRIDGE (Optional - For web-based GUI) -->
  <!-- Uncomment this section if you have rosbridge_server installed and want a web
GUI.
       Requires 'rosbridge_server' package to be installed:
       sudo apt install ros-noetic-rosbridge-server
  -->
  <!-- <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch" />
-->

  <!-- SECTION 4: RQT (Optional - for topic monitoring) -->
  <!-- Uncomment these to automatically launch rqt_graph or rqt_plot for real-time
topic visualization.
       Requires 'rqt_graph' and 'rqt_plot' packages:
       sudo apt install ros-noetic-rqt-graph ros-noetic-rqt-plot
  -->
  <!-- <node pkg="rqt_graph" type="rqt_graph" name="rqt_graph" /> -->
  <!-- <node pkg="rqt_plot" type="rqt_plot" name="rqt_plot" /> -->

  <!-- SECTION 5: RVIZ (Optional - for visualization) -->
  <!-- You may need to create or modify a .rviz configuration file -->
  <arg name="start_visualization" default="false" />
  <group if="$(arg start_visualization)">
    <!-- Replace 'sentience_display.rviz' with your actual RViz config file if you have one
-->
    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
sentience)/rviz/sentience_display.rviz" />
  </group>

  <!-- SECTION 6: ROS_WEB_SERVER (Optional - If you have a custom web server for
monitoring) -->
  <!-- If you have a separate web server node, configure it here -->
  <!-- <arg name="web_server_port" default="8080" />
  <node pkg="ros_web_server" type="web_server.py" name="ros_web_server"
output="screen">
    <param name="port" value="$(arg web_server_port)" />
  </node> -->

  <!-- SECTION 7: GAZEBO SIMULATION (Optional - for physical simulation) -->
```

```
  <!-- If you have a Gazebo simulation set up for your robot, you would include its launch file here.
       This typically involves loading a robot model and its environment.
  -->
  <!-- <include file="$(find YOUR_ROBOT_DESCRIPTION_PACKAGE)/launch/robot_in_gazebo.launch" /> -->

</launch>
```

8. Make Nodes Executable

Ensure all your Python node scripts (including mock_sensors.py) are executable:

```
chmod +x ~/catkin_ws/src/sentience/scripts/*.py
```

Running Sentience

1. **Open your first terminal and start your ROS Master:**
   ```
   roscore
   ```

2. Open your second terminal and start the Phi-2 Local Inference Server:
   Navigate to your llama.cpp directory (where you built it) and run the server command you used during installation:
   ```
   cd <path_to_your_llama.cpp_directory>
   ./server -m models/phi-2/phi-2.Q4_K_M.gguf -c 4096 --port 8000 --host 0.0.0.0
   ```

   (Ensure this server is fully initialized and running before proceeding.)
3. **Open your third terminal, source your ROS workspace, and launch the Sentience System:**
   ```
   source /opt/ros/noetic/setup.bash
   source ~/catkin_ws/devel/setup.bash
   roslaunch sentience sentience_system.launch
   ```

   You should see verbose output from all the launched nodes, indicating their initialization and operation.

Monitoring and Visualization

- rqt_graph: To visualize the ROS computational graph (nodes and topics), open a new terminal (and source your ROS workspace) and run:

rqt_graph

This helps verify that all nodes are connected as expected and topics are being published.

- rqt_plot / rostopic echo: To inspect the data flowing through specific topics, use rqt_plot for live plotting of numerical data or rostopic echo for raw message output.
  # Example: Plotting the priority score from the attention node
  rqt_plot /attention_state/priority_score

  # Example: Echoing the internal narrative messages
  rostopop echo /internal_narrative

  # Example: Echoing raw sensor data (watch for JSON content)
  rostopic echo /raw_sensor_data

Troubleshooting

- ImportError: No module named sentience.msg: This error means your custom .msg files haven't been built correctly into Python modules.
  - Solution: Re-check **Step 4: Defining and Building Custom ROS Messages** carefully. Ensure all .msg files are in sentience/msg/, package.xml and CMakeLists.txt are correctly updated with all message names, and you've run catkin_make successfully from your workspace root.
- Connection refused to LLM server:
  - Solution:
    - Verify the llama.cpp server (or equivalent) is running on the correct host and port (http://localhost:8000 by default, as configured in config.yaml).
    - Check if any firewall rules are blocking the connection on port 8000.
    - Ensure the llm_params.base_url parameter in your config.yaml is exactly correct.
    - If running in a virtual machine or Docker, ensure port forwarding is correctly set up.
- TimeoutError from LLM calls:
  - Solution:
    - The LLM inference might be too slow for the default timeout_seconds setting (30.0 seconds) in config.yaml. Try increasing this value in config.yaml (e.g., to 60.0 or 90.0).
    - Your machine might not have sufficient resources (CPU, RAM, or GPU if

using a GPU-accelerated llama.cpp build) to run Phi-2 efficiently. Consider using a smaller model quantization (e.g., Q4_K_S instead of Q4_K_M) or upgrading your hardware.
        - The max_tokens generated by the LLM might be too high. While nodes usually request reasonable lengths, an LLM misconfiguration could lead to very long responses that time out.
- Nodes not starting or crashing immediately:
    - Solution:
        - Check the ROS log level in your config.yaml and temporarily increase it to DEBUG (default_log_level: DEBUG) to get more verbose output on startup failures or exceptions.
        - Review the output in the terminal where you launched the nodes; Python tracebacks will be visible there.
        - Ensure all Python scripts have executable permissions (**Step 8**).
        - Verify that utils.py is correctly placed in sentience/scripts/.
- Nodes using std_msgs/String messages despite custom .msg files: This means that even if you created the .msg files, the Python environment where ROS is running the nodes hasn't found the generated custom message classes.
    - Solution: Double-check that you have sourced your ROS workspace's devel/setup.bash (or install/setup.bash for ROS 2) in *every new terminal* you open *before* running any roslaunch or rosrun commands. The message files need to be discoverable in the Python path.
- Database Errors (sqlite3.OperationalError):
    - Solution: Ensure the db_root_path in your config.yaml points to a writable directory (e.g., /tmp/sentience_db). The sentience package needs write permissions to create its SQLite databases there. If /tmp is not suitable, choose another path in your user's home directory.

Directory Structure

```
~/catkin_ws/src/sentience/
├── CMakeLists.txt
├── package.xml
├── msg/                    # Custom ROS message definitions (.msg files are here)
│   ├── ActionExecutionResult.msg
│   ├── ActionProposal.msg
│   ├── ... (all other .msg files)
├── scripts/                # Python ROS nodes and utility scripts
│   ├── attention_node.py
│   ├── bias_mitigation_node.py
```

```
│   ├── ... (all other .py nodes)
│   └── utils.py              # Shared utility functions (parse_ros_message_data,
load_config)
├── config/                   # Configuration files (YAML, JSON)
│   ├── config.yaml           # Main system configuration
│   └── ethical_principles.json # Ethical principles for Ethical Reasoning Node
└── launch/                   # ROS launch files
    └── sentience_system.launch # Main launch file to start all nodes
# (Optional ROS directories, if you extend the project)
├── srv/                      # ROS service definitions
├── action/                   # ROS action definitions
├── include/sentience/        # C++ header files
├── src/                      # C++ source files
├── rviz/                     # RViz configuration files
│   └── sentience_display.rviz
└── test/                     # Unit and integration tests
    └── ...
```

Contributing

If you wish to contribute to the Sentience project, please adhere to the following guidelines:

- Fork the repository.
- Create a new branch for your features or bug fixes.
- Follow ROS Python coding conventions and maintain consistency with existing code style.
- Ensure all new features are thoroughly tested (unit tests, integration tests).
- Update the README.md, .msg definitions, configuration files, and launch files as necessary.
- Submit a pull request with a clear description of changes.

License