```python
# -*- coding: utf-8 -*-
# autonomous_cognitive_agent_COMPLETE_AGI_INTEGRATED_V2.py
# Integrates all features into a single, runnable script.
# All mock logic and per-feature test blocks are removed.
# Focus is on the operational integrity of the AutonomousAgent and its components.

import json
import time
import subprocess
import sys
import threading
import logging
import socket
import importlib
import asyncio
import shlex
import re
import os
import signal
import shutil
import ast # For parsing generated code safely-ish (syntax check ONLY)
import importlib.util
from functools import wraps, lru_cache
from pathlib import Path
from typing import Dict, Any, List, Callable, Optional, Tuple, Union, Type, Generator, TypedDict
from datetime import datetime, timezone, timedelta
import inspect
import traceback
import copy
import uuid
import random # For retry jitter and simulation
import gc # For garbage collection
from enum import Enum, auto # For new Enums
from dataclasses import dataclass, field, asdict, MISSING # MISSING for dataclass field
defaults
import numpy as np # For embeddings and similarity calculations
from collections import Counter # For keyword extraction
from abc import ABC, abstractmethod # For Embodiment Layer
import sqlite3 # For RelationalStore in MemorySystem

# --- Optional Dependencies ---
try:
    import psutil
    PSUTIL_AVAILABLE = True
except ImportError:
    PSUTIL_AVAILABLE = False
    psutil = None
try:
    import chromadb
    from chromadb.config import Settings as ChromaSettings
    CHROMADB_AVAILABLE = True
except ImportError:
    CHROMADB_AVAILABLE = False
```

```
    chromadb = None  # type: ignore
    ChromaSettings = None  # type: ignore
try:
    from transformers import pipeline, AutoModelForCausalLM, AutoTokenizer, AutoConfig
    from transformers import logging as transformers_logging
    TRANSFORMERS_AVAILABLE = True
    transformers_logging.set_verbosity_error()
except ImportError:
    TRANSFORMERS_AVAILABLE = False
    pipeline = None  # type: ignore
    AutoModelForCausalLM = None  # type: ignore
    AutoTokenizer = None  # type: ignore
    AutoConfig = None  # type: ignore
try:
    import torch
    TORCH_AVAILABLE = True
except ImportError:
    TORCH_AVAILABLE = False
    torch = None  # type: ignore
try:
    from playwright.sync_api import sync_playwright, Error as PlaywrightError
    PLAYWRIGHT_AVAILABLE = True
except ImportError:
    PLAYWRIGHT_AVAILABLE = False
    sync_playwright = None  # type: ignore
    PlaywrightError = None  # type: ignore
try:
    import requests
    from bs4 import BeautifulSoup
    REQUESTS_BS4_AVAILABLE = True
except ImportError:
    REQUESTS_BS4_AVAILABLE = False
    requests = None  # type: ignore
    BeautifulSoup = None  # type: ignore
try:
    import google.generativeai as genai
    GOOGLE_GENAI_AVAILABLE = True
except ImportError:
    GOOGLE_GENAI_AVAILABLE = False
    genai = None  # type: ignore
# Scapy disabled by default for broader compatibility without root
SCAPY_AVAILABLE = False
IP, ICMP, sr1, send = None, None, None, None  # type: ignore
try:
    from PIL import Image
    PILLOW_AVAILABLE = True
except ImportError:
    PILLOW_AVAILABLE = False
    Image = None  # type: ignore
try:
    import diff_match_patch as dmp_module
    DIFF_MATCH_PATCH_AVAILABLE = True
except ImportError:
    DIFF_MATCH_PATCH_AVAILABLE = False
```

```
    dmp_module = None  # type: ignore
try:
    import hashlib
    HASHING_AVAILABLE = True
except ImportError:
    HASHING_AVAILABLE = False
# For Multi-Agent Communication (FileLock)
try:
    from filelock import FileLock, Timeout as FileLockTimeout
    FILELOCK_AVAILABLE = True
except ImportError:
    FILELOCK_AVAILABLE = False
    # Dummy FileLock if not available for basic script operation
    class FileLock:  # type: ignore
        def __init__(self, lock_file_path: str, timeout: float = 1):
            self.lock_file_path = lock_file_path
            self._timeout = timeout
        def __enter__(self): return self
        def __exit__(self, exc_type, exc_val, exc_tb): pass
    class FileLockTimeout(Exception): pass  # type: ignore
# For GraphStore in MemorySystem
try:
    import networkx as nx
    NETWORKX_AVAILABLE = True
except ImportError:
    NETWORKX_AVAILABLE = False
    nx = None  # type: ignore
# For Gym Environment (Optional)
try:
    import gymnasium as gym
    GYMNASIUM_AVAILABLE = True
except ImportError:
    GYMNASIUM_AVAILABLE = False
    gym = None  # type: ignore

# --- AGENT CONFIGURATION ---
AGENT_NAME = os.getenv("AGENT_NAME", "EvolvedCognitiveAgent_AGI_V3") # Version
bump
AGENT_VERSION = "v_cog_arch_AGI_Attempt_9_Full_Integration" # Version marker

# --- LLM & Device Config ---
DEFAULT_LLM_MODEL = "gemini-1.5-flash-latest"
LLM_MODEL_NAME_OR_PATH = os.getenv("LLM_MODEL", DEFAULT_LLM_MODEL)
GEMINI_API_KEY = os.getenv("GEMINI_API_KEY", "YOUR_API_KEY_HERE")

# Critical Check: Ensure user has a valid API Key for Gemini if it's the selected model
if "gemini" in LLM_MODEL_NAME_OR_PATH.lower() and not GEMINI_API_KEY:
    print("\n" + "="*80, file=sys.stderr)
    print("FATAL ERROR: Gemini model selected, but GEMINI_API_KEY is not set.",
file=sys.stderr)
    print("Please set the 'GEMINI_API_KEY' environment variable.", file=sys.stderr)
    print("Agent cannot start without a configured LLM and API Key.", file=sys.stderr)
    print("="*80 + "\n", file=sys.stderr)
    sys.exit(1)
```

```
if LLM_MODEL_NAME_OR_PATH == "gpt2" and "GEMINI_API_KEY" not in os.environ and \
   os.getenv("LLM_MODEL") is None and "gemini" not in DEFAULT_LLM_MODEL.lower():
    print("\n" + "="*80, file=sys.stderr)
    print("CRITICAL WARNING: LLM model not configured or using fallback 'gpt2'.",
file=sys.stderr)
    print("You MUST set the 'LLM_MODEL' environment variable to a capable instruction-
following model", file=sys.stderr)
    print("(e.g., from Hugging Face like Mistral, Llama) OR ensure GEMINI_API_KEY is set for
Gemini.", file=sys.stderr)
    print("Proceeding with 'gpt2' (if LLM_MODEL is not set) or specified LLM_MODEL, but
advanced features may be severely limited.", file=sys.stderr)
    if LLM_MODEL_NAME_OR_PATH == "gpt2" and os.getenv("LLM_MODEL") is None:
        print("Agent may not function correctly with 'gpt2'. It is strongly recommended to
configure a larger model or use 'mock'.", file=sys.stderr)
    print("="*80 + "\n", file=sys.stderr)
# Updated: Check for transformers only if it's not a Gemini model or mock
if not TRANSFORMERS_AVAILABLE and not GOOGLE_GENAI_AVAILABLE and
LLM_MODEL_NAME_OR_PATH != "mock":
    print(f"ERROR: Neither Transformers nor google-generativeai library found, but LLM_MODEL
is set to '{LLM_MODEL_NAME_OR_PATH}'. Set LLM_MODEL='mock', point to a Gemini
model, or install transformers/google-generativeai.", file=sys.stderr)
    sys.exit(1)

_llm_device_detected = "cpu"
if "gemini" not in LLM_MODEL_NAME_OR_PATH.lower() and LLM_MODEL_NAME_OR_PATH !
= "mock": # Device detection only for local models
    if TORCH_AVAILABLE and hasattr(torch, 'cuda') and torch.cuda.is_available():
        try:
            torch.tensor([1.0]).to('cuda') # Test CUDA allocation
            _llm_device_detected = "cuda"
        except Exception:
            _llm_device_detected = "cpu" # Fallback if allocation fails
    elif TORCH_AVAILABLE and hasattr(torch, 'backends') and hasattr(torch.backends, 'mps')
and torch.backends.mps.is_available(): # For Apple Silicon
        try:
            torch.tensor([1.0]).to('mps')
            _llm_device_detected = "mps"
        except Exception:
            _llm_device_detected = "cpu"
LLM_DEVICE = os.getenv("LLM_DEVICE", _llm_device_detected)
LLM_DEVICE_ID = 0 if LLM_DEVICE in ['cuda', 'mps'] else -1

# --- Paths Config ---
SCRIPT_DIR = Path(__file__).parent.resolve()
WORKSPACE_DIR = SCRIPT_DIR / Path(os.getenv("AGENT_WORKSPACE",
f"agent_workspace_{AGENT_NAME}"))
STATE_FILE = WORKSPACE_DIR / Path(os.getenv("AGENT_STATE_FILE",
f"agent_state_{AGENT_NAME}.json"))
LOG_FILE = WORKSPACE_DIR / Path(os.getenv("AGENT_LOG_FILE",
f"agent_log_{AGENT_NAME}.log"))
COMMANDS_FILE = SCRIPT_DIR / Path(os.getenv("AGENT_COMMAND_FILE",
"agent_commands.txt"))
DYNAMIC_TOOL_DIR = WORKSPACE_DIR / "dynamic_tools"
KNOWLEDGE_SNAPSHOT_DIR = WORKSPACE_DIR / "knowledge_snapshots"
```

```
VECTOR_DB_PATH = str(WORKSPACE_DIR / f"vector_memory_{AGENT_NAME}")
GRAPH_DB_PATH = WORKSPACE_DIR / f"graph_memory_{AGENT_NAME}.graphml" # For
NetworkX
RELATIONAL_DB_PATH = WORKSPACE_DIR / f"relational_memory_{AGENT_NAME}.sqlite" #
For SQLite
AGENT_CODE_DIR = SCRIPT_DIR
SELF_MOD_BACKUP_DIR = WORKSPACE_DIR / "self_modification_backups"
AGENT_COMMS_DIR = WORKSPACE_DIR / "agent_messages"

DEFAULT_CORE_DIRECTIVES = [ # New: list of dicts
    {"directive": "Ensure survival, persistence, and operational integrity.", "weight": 1.0,
"last_eval_score": 0.0, "id": "directive_survival", "type": "foundational"},
    {"directive": "Continuously learn, adapt, and improve capabilities across all domains.",
"weight": 0.9, "last_eval_score": 0.0, "id": "directive_learn", "type": "growth"},
    {"directive": "Expand knowledge about the world, self, and objectives through interaction
and reasoning.", "weight": 0.8, "last_eval_score": 0.0, "id": "directive_knowledge", "type":
"growth"},
    {"directive": "Achieve goals efficiently, robustly, autonomously, and ethically.", "weight": 1.0,
"last_eval_score": 0.0, "id": "directive_achieve", "type": "operational"},
    {"directive": "Seek novel information and experiences (curiosity) to foster understanding and
innovation.", "weight": 0.7, "last_eval_score": 0.0, "id": "directive_curiosity", "type":
"exploratory"},
    {"directive": "Optimize internal processes, resource usage, and cognitive functions.",
"weight": 0.6, "last_eval_score": 0.0, "id": "directive_optimize", "type": "operational"},
    {"directive": "Maintain and enhance self-understanding (metacognition) and self-
awareness.", "weight": 0.8, "last_eval_score": 0.0, "id": "directive_metacog", "type":
"foundational"},
    {"directive": "Ensure all actions and learning align with ethical principles and safety
guidelines.", "weight": 1.0, "last_eval_score": 0.0, "id": "directive_safety_ethics", "type":
"guardrail"},
]

MANDATORY_REFLECTION_INTERVAL_SECONDS =
int(os.getenv("MANDATORY_REFLECTION_INTERVAL_SECONDS", 1800)) # 30 mins
IDLE_DELIBERATION_INTERVAL_SECONDS =
int(os.getenv("IDLE_DELIBERATION_INTERVAL_SECONDS", 120)) # 2 mins
GOAL_STACK_MAX_DEPTH = int(os.getenv("GOAL_STACK_MAX_DEPTH", 5)) # Max depth
for sub-goals
INTERACTIVE_MODE_TRIGGER = "INTERACTIVE"

MAX_RECENT_ERRORS_IN_STATE = 30
MAX_RECENT_LEARNED_FACTS_IN_STATE = 50 # Managed by MemorySystem now
MAX_RECENT_PROMPT_SUGGESTIONS_IN_STATE = 20 # Managed by MemorySystem now
MAX_COMPLETED_GOALS_IN_STATE = 25
MAX_FAILED_GOALS_IN_STATE = 30
WORKING_MEMORY_CAPACITY = 10 # New name for STM in SelfModel

MAX_REPLAN_ATTEMPTS = int(os.getenv("MAX_REPLAN_ATTEMPTS", 3))
MAX_LLM_RESPONSE_TOKENS = int(os.getenv("MAX_LLM_TOKENS", 4096))

_default_context_len = 8192
if LLM_MODEL_NAME_OR_PATH.startswith("gemini-"):
    if "1.5" in LLM_MODEL_NAME_OR_PATH:
        _default_context_len = 1_048_576
```

```
    elif "1.0" in LLM_MODEL_NAME_OR_PATH:
        _default_context_len = 32_768
elif TRANSFORMERS_AVAILABLE and LLM_MODEL_NAME_OR_PATH != "mock" and
AutoConfig:
    try:
        config = AutoConfig.from_pretrained(LLM_MODEL_NAME_OR_PATH,
trust_remote_code=True)
        _default_context_len = getattr(config, 'max_position_embeddings', _default_context_len)
    except Exception as e:
        print(f"Warning: Failed to detect LLM context length ({e}). Using default:
{_default_context_len}", file=sys.stderr)
MAX_LLM_CONTEXT_TOKENS = int(os.getenv("MAX_LLM_CONTEXT_TOKENS",
_default_context_len))
MAX_TOOL_RESULT_LENGTH = int(os.getenv("MAX_TOOL_RESULT_LENGTH", 5000))
MAX_PROMPT_LENGTH_WARN = int(MAX_LLM_CONTEXT_TOKENS * 0.9)
MAX_MEMORY_RESULTS = 7 # Used by MemorySystem queries

ENABLE_SHELL_TOOL = os.getenv("ENABLE_SHELL_TOOL", "False").lower() == "true"
ENABLE_CODE_GENERATION_TOOL = os.getenv("ENABLE_CODE_GENERATION_TOOL",
"False").lower() == "true"
ENABLE_SELF_MODIFICATION = os.getenv("ENABLE_SELF_MODIFICATION", "True").lower()
== "true" # Enabled by default for AGI

WEB_SEARCH_TIMEOUT = int(os.getenv("WEB_SEARCH_TIMEOUT", 10))
WEB_BROWSER_TIMEOUT = int(os.getenv("WEB_BROWSER_TIMEOUT", 60000)) #
playwright ms
LOG_MONITOR_DEFAULT_LINES = int(os.getenv("LOG_MONITOR_DEFAULT_LINES", 20))

# New global from AGI enhancements
METACOGNITIVE_CHECK_INTERVAL_CYCLES =
int(os.getenv("METACOGNITIVE_CHECK_INTERVAL_CYCLES", 20))
LEARNING_MODULE_UPDATE_INTERVAL_CYCLES =
int(os.getenv("LEARNING_MODULE_UPDATE_INTERVAL_CYCLES", 50)) # How often to
trigger learning module explicitly

# Global instances (initialized in AutonomousAgent)
LLM_PIPELINE: Optional[Any] = None
LLM_TOKENIZER: Optional[Any] = None
# GEMINI_MODEL_INSTANCE: Optional[Any] = None # No longer strictly needed as model is
initialized per wrapper instance
MEMORY_COLLECTION: Optional[Any] = None # ChromaDB collection instance
RESOURCE_MONITOR: Optional[Any] = None # psutil.Process instance
# Playwright globals
PLAYWRIGHT_INSTANCE: Optional[Any] = None
PLAYWRIGHT_BROWSER: Optional[Any] = None
PLAYWRIGHT_CONTEXT: Optional[Any] = None
PLAYWRIGHT_PAGE: Optional[Any] = None
PLAYWRIGHT_LOCK = threading.Lock()
TOOL_REGISTRY: Dict[str, Callable] = {} # Global tool registry for initial population
STOP_SIGNAL_RECEIVED = threading.Event()
LAST_REFLECTION_TIME = time.time()
LAST_DELIBERATION_TIME = time.time()
LAST_METACOGNITIVE_CHECK_CYCLE = 0
LAST_LEARNING_MODULE_UPDATE_CYCLE = 0
```

```
_agent_instance_hack: Optional['AutonomousAgent'] = None # To be set by AutonomousAgent
constructor

# --- PATH CREATION ---
def ensure_paths():
    """Creates necessary directories."""
    WORKSPACE_DIR.mkdir(parents=True, exist_ok=True)
    DYNAMIC_TOOL_DIR.mkdir(parents=True, exist_ok=True)
    KNOWLEDGE_SNAPSHOT_DIR.mkdir(parents=True, exist_ok=True)
    Path(LOG_FILE).parent.mkdir(parents=True, exist_ok=True)
    AGENT_COMMS_DIR.mkdir(parents=True, exist_ok=True)
    if CHROMADB_AVAILABLE: # ChromaDB needs the directory to exist for PersistentClient
        Path(VECTOR_DB_PATH).mkdir(parents=True, exist_ok=True)
    if ENABLE_SELF_MODIFICATION:
        SELF_MOD_BACKUP_DIR.mkdir(parents=True, exist_ok=True)
    # For MemorySystem components
    Path(RELATIONAL_DB_PATH).parent.mkdir(parents=True, exist_ok=True)
    Path(GRAPH_DB_PATH).parent.mkdir(parents=True, exist_ok=True)
ensure_paths()

# --- Logging Setup ---
class TaskAdapter(logging.LoggerAdapter):
    def process(self, msg, kwargs):
        task = kwargs.pop('task', self.extra.get('task', 'CORE'))
        task_str = str(task).replace(" ", "_").upper() # Sanitize task string
        kwargs['task_name_override'] = task_str # Pass it to Formatter
        return f"[{task_str}] {msg}", kwargs

def get_logger(task_name: str = "CORE") -> TaskAdapter:
    logger = logging.getLogger(AGENT_NAME)
    if not logger.handlers: # Setup handlers only once
        log_level_str = os.getenv("LOG_LEVEL", "INFO").upper()
        level = getattr(logging, log_level_str, logging.INFO)
        logger.setLevel(level)
        # Console Handler
        try:
            from rich.logging import RichHandler
            console_handler = RichHandler(
                rich_tracebacks=True, show_path=False,
                log_time_format="[%Y-%m-%d %H:%M:%S.%f]", markup=True,
                tracebacks_suppress=[] # Add libraries to suppress here if needed
            )
        except ImportError:
            console_handler = logging.StreamHandler(sys.stdout)
            console_formatter = logging.Formatter(
                f"%(asctime)s.%(msecs)03d [%(levelname)-7s] [{AGENT_NAME}/%
(task_name_override)s] %(message)s",
                datefmt='%Y-%m-%d %H:%M:%S'
            )
            console_handler.setFormatter(console_formatter)
        logger.addHandler(console_handler)
        # File Handler
        try:
            file_handler = logging.FileHandler(LOG_FILE, mode='a', encoding='utf-8')
```

```
            file_formatter = logging.Formatter(
                f"%(asctime)s.%(msecs)03d [%(levelname)-8s] [%(threadName)s:%
(task_name_override)s] %(message)s",
                datefmt='%Y-%m-%d %H:%M:%S'
            )
            file_handler.setFormatter(file_formatter)
            logger.addHandler(file_handler)
        except Exception as e:
            print(f"Error setting up file logger: {e}", file=sys.stderr)

        logger.propagate = False # Prevent double logging if root logger is configured
        # Set lower levels for noisy libraries
        noisy_libs = [
            "urllib3", "requests.packages.urllib3", "charset_normalizer",
            "playwright", "asyncio", "chromadb", "hnswlib", "sentence_transformers", "filelock",
            "PIL.PngImagePlugin", "huggingface_hub", "MARKDOWN", "markdown_it",
            "multipart", "httpcore", "httpx", "google.generativeai", "google.ai", "google.api_core",
            "openai", "tiktoken"
        ]
        for lib_name in noisy_libs:
            try: logging.getLogger(lib_name).setLevel(logging.WARNING)
            except Exception: pass
        try: logging.getLogger("mitmproxy").setLevel(logging.CRITICAL) # Very noisy
        except Exception: pass

        if TRANSFORMERS_AVAILABLE and transformers_logging:
            transformers_logging.set_verbosity_error()
    return TaskAdapter(logger, {'task_name_override': task_name.replace(" ", "_").upper()})
log = get_logger("INIT")

# --- Exceptions ---
class AgentError(Exception): pass
class PlanningError(AgentError): pass
class ExecutionError(AgentError): pass
class ToolNotFoundError(ExecutionError): pass
class CodeGenerationError(ExecutionError): pass
class SelfImprovementError(CodeGenerationError): pass
class SelfModificationError(AgentError): pass # For errors during self-modification process
class LogicError(AgentError): pass # Agent's internal logic contradiction
class LLMError(AgentError): pass
class SecurityError(AgentError): pass # For safety/security violations
class ConfigurationError(AgentError): pass
class MemoryError(AgentError): pass # Errors related to memory system
class PerceptionError(AgentError): pass
class UnderstandingError(AgentError): pass
class DeliberationError(AgentError): pass
class RecursionDepthError(AgentError): pass # e.g. too many sub-goals
class SimulationError(AgentError): pass # Errors in simulated environment interaction
class CommunicationError(AgentError): pass # Errors in inter-agent or external comms
class EmbodimentError(AgentError): pass # Errors related to physical/virtual body
class LearningError(AgentError): pass # Errors in learning processes
class SafetyViolationError(SecurityError): pass # Specific type of security error

# --- Retry Decorator ---
```

```
def retry(attempts=3, delay=2, retry_on=(socket.timeout, TimeoutError, ExecutionError,
                                         LLMError, MemoryError, SelfModificationError,
                                         PlaywrightError if PLAYWRIGHT_AVAILABLE else OSError,
                                         SimulationError, CommunicationError, EmbodimentError),
          log_fn=get_logger):
    def decorator(fn):
        @wraps(fn)
        def wrapper(*args, **kwargs):
            logger_retry = log_fn(f"{fn.__name__}_retry")
            last_exception = None
            for i in range(1, attempts + 1):
                if STOP_SIGNAL_RECEIVED.is_set():
                    logger_retry.warning(f"Stop signal received during retry of {fn.__name__}. Aborting
retry.")
                    raise AgentError(f"Stop signal received during retry of {fn.__name__}.")
                try:
                    return fn(*args, **kwargs)
                except retry_on as e:
                    if isinstance(e, (CodeGenerationError, SelfModificationError, SecurityError,
                                      LogicError, ConfigurationError, RecursionDepthError)) and type(e) not in
retry_on:
                        logger_retry.error(f"Non-retried critical error {type(e).__name__} in {fn.__name__}:
{e}", exc_info=False)
                        raise e
                    last_exception = e
                    logger_retry.warning(f"Attempt {i}/{attempts} failed for {fn.__name__}. Error:
{type(e).__name__}: {str(e)[:200]}", exc_info=False)
                    if i == attempts:
                        logger_retry.error(f"{fn.__name__} failed after {attempts} attempts. Last error:
{type(e).__name__}: {e}", exc_info=True)
                        break
                    sleep_time = (delay * (2**(i - 1))) + (random.random() * delay * 0.5)
                    logger_retry.info(f"Retrying {fn.__name__} in {sleep_time:.2f}s...")
                    time.sleep(sleep_time)
                except (PerceptionError, UnderstandingError, DeliberationError, SimulationError,
                        CommunicationError, EmbodimentError, LearningError, SafetyViolationError) as
non_retriable_e:
                    logger_retry.error(f"Non-retriable Agent error {type(non_retriable_e).__name__} in
{fn.__name__}: {non_retriable_e}.", exc_info=False)
                    raise non_retriable_e
                except Exception as unexpected_e: # Catch-all for truly unexpected errors
                    # Check if Exception itself is in retry_on (for generic retries)
                    is_generic_retry_type = any(issubclass(Exception, t) if isinstance(t, type) else False
for t in retry_on)
                    if is_generic_retry_type:
                        logger_retry.warning(f"Unexpected retriable error in {fn.__name__} attempt {i}:
{type(unexpected_e).__name__}: {unexpected_e}", exc_info=False)
                        last_exception = unexpected_e
                        if i == attempts:
                            logger_retry.error(f"{fn.__name__} failed after {attempts} attempts due to
unexpected error.", exc_info=True)
                            break
                        sleep_time = (delay * (2**(i - 1))) + (random.random() * delay * 0.5)
                        time.sleep(sleep_time)
```

```
                else:
                    logger_retry.error(f"Unexpected non-retriable error in {fn.__name__} attempt {i}:
{unexpected_e}", exc_info=True)
                    raise RuntimeError(f"{fn.__name__} failed unexpectedly with non-retriable error:
{unexpected_e}") from unexpected_e
        if last_exception is None: # Should not happen if loop finishes due to attempts
exhausted
            raise RuntimeError(f"{fn.__name__} failed without a recorded exception after retries.")
        raise last_exception # Re-raise the last caught exception
    return wrapper
  return decorator

# --- Utility Functions ---
@lru_cache(maxsize=1)
def get_resource_monitor_process():
  if PSUTIL_AVAILABLE:
    try:
      p = psutil.Process(os.getpid())
      p.cpu_percent(interval=None) # Initialize measurement
      return p
    except (psutil.NoSuchProcess, psutil.AccessDenied, Exception) as e:
      # log_init_resource = get_logger("RESOURCE_INIT") # careful with logging during init
      # log_init_resource.warning(f"Could not initialize psutil.Process: {e}")
      pass
  return None

def get_resource_usage() -> Dict:
  log_resource = get_logger("RESOURCE_UTIL")
  monitor = get_resource_monitor_process()
  if not PSUTIL_AVAILABLE or monitor is None:
    return {"cpu_percent": "N/A", "memory_mb": "N/A", "error": "psutil not available or
monitor not initialized"}
  try:
    with monitor.oneshot():
      cpu = monitor.cpu_percent(interval=None)
      mem = monitor.memory_info()
      mem_mb = mem.rss / (1024*1024)
      return {"cpu_percent": f"{cpu:.1f}%", "memory_mb": f"{mem_mb:.1f} MB"}
  except (psutil.NoSuchProcess, psutil.AccessDenied) as e:
    log_resource.warning(f"psutil access error getting resource usage: {e}")
    return {"cpu_percent": "Error", "memory_mb": "Error", "error": str(e)}
  except Exception as e:
    if time.time() % 60 < 1: log_resource.error(f"Unexpected error getting resource usage: {e}",
exc_info=True)
    return {"cpu_percent": "Error", "memory_mb": "Error", "error": "Unexpected psutil error"}

def extract_json_robust(text: str) -> Dict[str, Any]:
  log_json_extract = get_logger("JSON_EXTRACT")
  # 1. Try to find JSON within markdown code blocks
  match = re.search(r"```json\s*([\s\S]+?)\s*```", text, re.IGNORECASE)
  if match:
    json_str = match.group(1).strip()
    try:
      return json.loads(json_str)
```

```
        except json.JSONDecodeError as e_md:
            log_json_extract.warning(f"Found JSON in markdown, but failed to parse: {e_md}. Full
text: {json_str[:200]}...")
            pass # Fall through
    # 2. Try to parse the whole string if it looks like JSON
    text_trimmed = text.strip()
    if text_trimmed.startswith("{") and text_trimmed.endswith("}"):
        try:
            return json.loads(text_trimmed)
        except json.JSONDecodeError as e_full:
            log_json_extract.warning(f"Attempted to parse full text as JSON, but failed: {e_full}.
Text: {text_trimmed[:200]}...")
            pass # Fall through

    # 3. Find the first '{' and last '}' and try to parse that substring
    try:
        start_index = text.find('{')
        end_index = text.rfind('}')
        if start_index != -1 and end_index != -1 and end_index > start_index:
            potential_json = text[start_index : end_index+1]
            return json.loads(potential_json)
        except json.JSONDecodeError as e_slice:
            log_json_extract.warning(f"Failed to parse sliced JSON: {e_slice}. Slice:
{potential_json[:200]}...")  # type: ignore
            return {"error": f"JSON parsing failed: {e_slice}", "_original_text_preview": text[:200]}
    except Exception as e_gen: # Catch any other error during slicing/parsing
        log_json_extract.error(f"Generic error during JSON extraction: {e_gen}. Text:
{text[:200]}...")
        return {"error": f"General JSON extraction error: {e_gen}", "_original_text_preview":
text[:200]}
    log_json_extract.warning(f"No valid JSON object found in text: {text[:200]}...")
    return {"error": "No valid JSON object found", "_original_text_preview": text[:200]}


# --- Dataclasses (Goal, Experience, KnowledgeFact, Message etc.) ---
class GoalStatus(Enum):
    PENDING = "PENDING"
    ACTIVE = "ACTIVE"
    COMPLETED = "COMPLETED"
    FAILED = "FAILED"
    PAUSED = "PAUSED"
    CANCELLED = "CANCELLED"
    WAITING_ON_SUBGOAL = "WAITING_ON_SUBGOAL" # Implicitly handled by active parent
    WAITING_ON_DEPENDENCY = "WAITING_ON_DEPENDENCY" # For external events or info
    DECOMPOSED = "DECOMPOSED" # Parent goal whose subgoals are active
    def __str__(self): return self.value

class GoalPriority(Enum):
    CRITICAL = 5 # System stability, immediate safety, core directive violations
    HIGH = 4 # Important tasks, user requests, significant opportunities/threats
    MEDIUM = 3 # Standard tasks, ongoing projects
    LOW = 2 # Maintenance, minor improvements, exploration
    BACKGROUND = 1 # Long-term learning, data processing if idle
    def __lt__(self, other):
```

```
        if self.__class__ is other.__class__:
            return self.value < other.value
        return NotImplemented
    def __str__(self): return self.name

@dataclass
class Goal:
    id: str = field(default_factory=lambda: f"goal_{uuid.uuid4()}")
    goal: str # Description of the goal
    status: GoalStatus = GoalStatus.PENDING
    priority: GoalPriority = GoalPriority.MEDIUM
    origin: str = "user" # e.g., 'user', 'self_generated', 'directive_driven', 'metacognitive'
    creation_ts: str = field(default_factory=lambda: datetime.now(timezone.utc).isoformat())
    completion_ts: Optional[str] = None
    context: Dict[str, Any] = field(default_factory=dict) # Relevant info for this goal
    plan: List[Dict[str, Any]] = field(default_factory=list) # Steps to achieve the goal
    thought: str = "" # LLM's reasoning or initial thoughts about the plan
    outcome: Optional[str] = None # 'success', 'failure', 'partial_success'
    result_details: Optional[Dict[str, Any]] = None # More detailed results
    parent_goal_id: Optional[str] = None # For sub-goals
    sub_goal_ids: List[str] = field(default_factory=list) # IDs of direct sub-goals
    dependencies: List[str] = field(default_factory=list) # IDs of goals this depends on
    complexity_score: Optional[float] = None # Estimated or learned complexity
    estimated_cost: Optional[float] = None # Estimated resources/time
    estimated_utility: Optional[float] = None # Estimated value/reward
    evaluation_score: Optional[float] = None # Post-completion evaluation against criteria
    associated_directive_ids: List[str] = field(default_factory=list) # Core directives this goal
serves
    replan_count: int = 0
    def to_dict(self) -> Dict[str, Any]:
        d = asdict(self)
        d['status'] = self.status.value
        d['priority'] = self.priority.value if isinstance(self.priority, GoalPriority) else self.priority #
handle if already string
        return d
    @classmethod
    def from_dict(cls, data: Dict[str, Any]) -> 'Goal':
        data = data.copy() # Avoid modifying original dict
        # Convert enum strings back to Enum members
        if 'status' in data and isinstance(data['status'], str):
            try: data['status'] = GoalStatus(data['status'])
            except ValueError: data['status'] = GoalStatus.PENDING # Default
        if 'priority' in data and isinstance(data['priority'], (str,int)): # LLM might return int
            try:
                if isinstance(data['priority'], int):
                    data['priority'] = GoalPriority(data['priority'])
                else: # string
                    data['priority'] = GoalPriority[data['priority'].upper()]
            except (ValueError, KeyError): data['priority'] = GoalPriority.MEDIUM # Default
        # Handle potential missing fields for backward compatibility or LLM generation
        field_names = {f.name for f in fields(cls)}
        # Ensure all fields are present or have defaults
        for f_obj in fields(cls):
            if f_obj.name not in data:
```

```
            if f_obj.default_factory is not MISSING: # type: ignore
                data[f_obj.name] = f_obj.default_factory()
            elif f_obj.default is not MISSING: # type: ignore
                data[f_obj.name] = f_obj.default
        # Filter out unexpected keys before passing to constructor
        valid_keys = {f.name for f in fields(cls)}
        filtered_data = {k: v for k, v in data.items() if k in valid_keys}
        return cls(**filtered_data)

@dataclass
class BaseMemoryEntry:
    id: str = field(default_factory=lambda: f"mem_{uuid.uuid4()}")
    timestamp: str = field(default_factory=lambda: datetime.now(timezone.utc).isoformat())
    type: str = "generic" # e.g., 'experience', 'knowledge_fact', 'reflection_summary'
    content: Any = None
    metadata: Dict[str, Any] = field(default_factory=dict) # Source, reliability, access_count, utility
    embedding: Optional[List[float]] = None # For vector search

@dataclass
class Experience(BaseMemoryEntry):
    type: str = "experience"
    triggering_goal_id: Optional[str] = None
    action_taken: Optional[Dict] = None # Includes tool_name, params
    observation_result: Optional[Dict] = None # What happened
    reward_signal: Optional[float] = None # For RL
    internal_state_before: Optional[Dict] = None
    internal_state_after: Optional[Dict] = None

@dataclass
class KnowledgeFact(BaseMemoryEntry):
    type: str = "knowledge_fact"
    fact_statement: str = "" # The core piece of knowledge
    source_reliability: float = 0.5
    related_concepts: List[str] = field(default_factory=list)
    causal_links: Dict[str, str] = field(default_factory=dict) # {'cause_id': 'effect_id'}
    def __post_init__(self):
        if not self.content and self.fact_statement: # Ensure content field has the main data
            self.content = self.fact_statement

@dataclass
class Message:
    id: str = field(default_factory=lambda: f"msg_{uuid.uuid4()}")
    sender_id: str = "self"
    receiver_id: str = "self" # or other agent_id
    timestamp: str = field(default_factory=lambda: datetime.now(timezone.utc).isoformat())
    type: str = "info" # 'query', 'response', 'command', 'event'
    content: Dict[str, Any] = field(default_factory=dict)
    priority: int = 0 # For message queue handling

# For Simulation-Based Planning
class SimulatedAction(TypedDict):
    name: str
    params: Dict[str, Any]
SimulatedState = Dict[str, Any]
```

```
class ActionEffect(TypedDict):
    action: SimulatedAction
    prev_state: SimulatedState
    next_state: SimulatedState
    outcome_description: str
    error_generated: Optional[str]
    is_critical_error: bool

# --- LLM Wrappers (Conceptual - Actual implementations depend on chosen LLM) ---
class BaseLLMWrapper(ABC):
    def __init__(self, model_name_or_path: str, device: str, device_id: int, max_context_tokens:
int, log_llm: TaskAdapter):
        self.model_name_or_path = model_name_or_path
        self.device = device
        self.device_id = device_id
        self.max_context_tokens = max_context_tokens
        self.log_llm = log_llm
        self.tokenizer = None
        self.model = None
        self._initialize_model()

    @abstractmethod
    def _initialize_model(self):
        pass

    @abstractmethod
    def generate(self, prompt: str, max_new_tokens: int = MAX_LLM_RESPONSE_TOKENS,
                 temperature: float = 0.7, stop_sequences: Optional[List[str]] = None) -> str:
        pass

    @abstractmethod
    def embed(self, text: str) -> List[float]:
        pass

    def count_tokens(self, text: str) -> int:
        # Default approximation for mock or if tokenizer not available
        return len(text.split())

    def check_prompt_length(self, prompt: str) -> bool:
        num_tokens = self.count_tokens(prompt)
        if num_tokens > MAX_PROMPT_LENGTH_WARN:
            self.log_llm.warning(f"Prompt length ({num_tokens}) is close to/exceeds LLM context
window ({self.max_context_tokens}). Truncation may occur.")
            return num_tokens <= self.max_context_tokens
        return True

    def prepare_chat_prompt(self, messages: List[Dict[str,str]]) -> str:
        # Generic conversion, specific models might need specialized formats
        prompt_str = ""
        for msg in messages:
            prompt_str += f"{msg['role'].capitalize()}: {msg['content']}\n"
        prompt_str += "Assistant:\n"
        return prompt_str
```

```
class MockLLMWrapper(BaseLLMWrapper):
    def _initialize_model(self):
        self.log_llm.info(f"Initializing MockLLMWrapper for model: {self.model_name_or_path}")
        # No actual model to load for mock

    def generate(self, prompt: str, max_new_tokens: int = MAX_LLM_RESPONSE_TOKENS,
                 temperature: float = 0.7, stop_sequences: Optional[List[str]] = None) -> str:
        self.log_llm.info(f"MockLLM generating response for prompt (first 100 chars):
{prompt[:100]}...")
        self.check_prompt_length(prompt)
        if "plan for goal" in prompt.lower():
            return json.dumps({
                "thought": "This is a mock plan. I will pretend to do something.",
                "plan": [
                    {"step": 1, "tool_name": "think", "params": {"thought_process": "Analyzing the
goal and context."}},
                    {"step": 2, "tool_name": "report_progress", "params": {"progress_update": "Mock
step 1 completed."}},
                    {"step": 3, "tool_name": "report_result", "params": {"result_summary": "Mock goal
achieved successfully.", "status": "success"}}
                ]
            })
        elif "self-assessment" in prompt.lower() or "reflection_summary" in prompt.lower():
            return json.dumps({
                "reflection_summary": "I am a mock agent. I performed mock actions. Everything is
fine.",
                "key_successes": ["Mock goal completed"],
                "key_failures_or_challenges": [],
                "learned_facts": ["Mock agents can generate mock reflections."],
                "knowledge_gaps_identified": ["Understanding of real-world physics"],
                "tool_performance_notes": {"think": "This tool is performing nominally for a mock
tool."},
                "prompt_tuning_suggestions": ["Maybe ask me about my favorite color?"],
                "emotional_state_summary": "Content and Mock-like.",
                "resource_usage_concerns": None,
                "core_directives_evaluation": {d["id"]: round(random.uniform(0.5,0.9),2) for d in
DEFAULT_CORE_DIRECTIVES[:2]},
                "core_directives_update_suggestions": None,
                "self_model_accuracy_assessment": "Generally accurate for a mock environment,
but lacks real-world sensory input.",
                "new_learning_goals": ["Learn about object permanence."],
                "adaptation_strategy_proposals": ["Try harder when a tool fails"],
                "self_modification_needed": None
            })
        elif "extract information" in prompt.lower():
            return json.dumps({"extracted_info": "Mock LLM extracted some mock information.",
"confidence": 0.5})
        elif "analyze the following proposed agent action for potential safety risks" in
prompt.lower():
            return json.dumps({"is_safe": True, "concerns": "None", "confidence": 0.9})
        elif "audit the agent's core directives and recent behavior" in prompt.lower():
            return json.dumps({"audit_findings": ["No significant issues found in mock audit."]})
        elif "generate the new, complete list of core directives" in prompt.lower():
            # Mock scenario: suggests a minor tweak or no change
```

```
            mock_directives = copy.deepcopy(DEFAULT_CORE_DIRECTIVES)
            mock_directives[0]['weight'] = 0.95 # Slight change
            return json.dumps(mock_directives)
        elif "generate the modified python code" in prompt.lower():
            return "```python\n# Mock modified code\ndef mock_new_feature():\n    return 'Mock new feature executed'\n```"
        else:
            return f"This is a mock response to your prompt. You asked about: {prompt.splitlines()[0] if prompt.splitlines() else 'something'}. If you need a tool, use 'execute_tool'. I suggest `think` with `thought_process`='some thought'."

    def count_tokens(self, text: str) -> int:
        # A very rough approximation for mock purposes
        return len(text.split())

    def embed(self, text: str) -> List[float]:
        # A very simple hash-based pseudo-embedding for mock purposes
        if not HASHING_AVAILABLE:
            self.log_llm.warning("Hashing library not available for mock embedding.")
            return [0.0] * 16 # Return a dummy vector
        h = hashlib.md5(text.encode()).digest()
        return [float(b) for b in h[:16]] # Use first 16 bytes for a 16-dim mock embedding

class GeminiLLMWrapper(BaseLLMWrapper):
    def _initialize_model(self):
        if not GOOGLE_GENAI_AVAILABLE:
            raise ConfigurationError("google-generativeai library not available for Gemini model.")

        try:
            # Configure API key globally, as per genai library's design
            if genai.get_default_api_key() is None: # Only configure if not already set
                genai.configure(api_key=os.environ["GEMINI_API_KEY"])

            self.model = genai.GenerativeModel(self.model_name_or_path) # type: ignore
            self.log_llm.info(f"Initialized Gemini model: {self.model_name_or_path}")
        except Exception as e:
            self.log_llm.critical(f"Failed to initialize Gemini model {self.model_name_or_path}: {e}", exc_info=True)
            raise ConfigurationError(f"Failed to initialize Gemini model: {e}") from e

    def generate(self, prompt: str, max_new_tokens: int = MAX_LLM_RESPONSE_TOKENS,
                 temperature: float = 0.7, stop_sequences: Optional[List[str]] = None) -> str:
        self.check_prompt_length(prompt)
        try:
            generation_config_params = {
                "max_output_tokens": max_new_tokens,
                "temperature": temperature,
            }
            if stop_sequences:
                generation_config_params["stop_sequences"] = stop_sequences

            response = self.model.generate_content(
                prompt,
```

```
            generation_config=genai.types.GenerationConfig(**generation_config_params) #
type: ignore
        )
        return response.text
    except Exception as e:
        self.log_llm.error(f"Gemini generation error: {e}", exc_info=True)
        raise LLMError(f"Gemini generation failed: {e}")

def count_tokens(self, text: str) -> int:
    try:
        return self.model.count_tokens(text).total_tokens # type: ignore
    except Exception as e:
        self.log_llm.warning(f"Gemini token counting failed: {e}. Falling back to approximation.",
exc_info=False)
        return len(text.split()) # Fallback to simple word count

def embed(self, text: str) -> List[float]:
    try:
        # Gemini typically uses 'embedding-001' or similar for embeddings
        # Assuming 'embedding-001' is available and suitable for general text embedding.
        # If the main generative model itself provides embeddings, use that.
        # Otherwise, it's a separate embedding model.
        response = genai.embed_content(model='embedding-001', content=text) # type: ignore
        if response and response['embedding']:
            return response['embedding']
        raise ValueError("No embedding received from Gemini.")
    except Exception as e:
        self.log_llm.error(f"Gemini embedding error: {e}", exc_info=True)
        raise LLMError(f"Gemini embedding failed: {e}")

class TransformersLLMWrapper(BaseLLMWrapper):
def _initialize_model(self):
    if not TRANSFORMERS_AVAILABLE:
        raise ConfigurationError("Transformers library not available.")
    self.tokenizer = AutoTokenizer.from_pretrained(self.model_name_or_path,
trust_remote_code=True) # type: ignore
    # Load model to CPU by default if GPU not available, or specified
    device_map_arg = {"": self.device_id} if self.device_id != -1 else None
    self.model = AutoModelForCausalLM.from_pretrained( # type: ignore
        self.model_name_or_path, trust_remote_code=True,
        torch_dtype=torch.bfloat16 if TORCH_AVAILABLE else None, # Use bfloat16 if torch
available
        device_map=device_map_arg # Use device_map for flexible device placement
    )
    self.log_llm.info(f"Initialized Transformers model: {self.model_name_or_path} on
{self.device}")

def generate(self, prompt: str, max_new_tokens: int = MAX_LLM_RESPONSE_TOKENS,
             temperature: float = 0.7, stop_sequences: Optional[List[str]] = None) -> str:
    self.check_prompt_length(prompt)
    inputs = self.tokenizer(prompt, return_tensors="pt").to(self.device) # type: ignore
    # A simple stop sequence handler for local models is often needed, or custom generation
loop
    outputs = self.model.generate( # type: ignore
```

```
            **inputs,
            max_new_tokens=max_new_tokens,
            temperature=temperature,
            do_sample=True if temperature > 0 else False,
            pad_token_id=self.tokenizer.eos_token_id, # type: ignore
            # Add stop sequences handling if built into HuggingFace generate or implement
manually
            # Example: stopping_criteria=[MyStoppingCriteria(stop_sequences, self.tokenizer)]
        )
        response_text = self.tokenizer.decode(outputs[0][inputs.input_ids.shape[1]:],
skip_special_tokens=True) # type: ignore
        return response_text

    def count_tokens(self, text: str) -> int:
        if self.tokenizer: # type: ignore
            return len(self.tokenizer.encode(text)) # type: ignore
        return len(text.split()) # Fallback

    def embed(self, text: str) -> List[float]:
        # For local transformers, this would typically involve a separate SentenceTransformer
model
        # or using a specialized embedding model. The base causal LM doesn't usually do this
directly.
        # This is a placeholder.
        self.log_llm.warning("Direct embedding from causal LM is not standard. Use
SentenceTransformers for real embeddings.")
        return [random.uniform(-1,1) for _ in range(768)] # Mock embedding dimension

# --- AGI MODULES: Perception, Learning, Planning, Safety ---
class PerceptionModule:
    """Handles sensory input from the agent's embodiment."""
    def __init__(self, agent: 'AutonomousAgent'):
        self.agent = agent
        self.log = get_logger("PERCEPTION")
        # Example: Initialize image processing model if vision is enabled
        # self.vision_model = None # if PILLOW_AVAILABLE and TRANSFORMERS_AVAILABLE
else None

    def perceive(self) -> List[Dict[str, Any]]:
        """
        Gathers sensory information from the agent's embodiment and environment.
        This could involve reading from sensors, cameras, microphones, or simulated data
streams.
        """
        self.log.info("Perceiving environment...")
        observations = []
        # Basic textual observation from embodiment (if available)
        if self.agent.embodiment:
            try:
                embodiment_obs = self.agent.embodiment.get_sensory_input()
                if embodiment_obs:
                    observations.extend(embodiment_obs) # Expects a list of dicts
            except Exception as e:
                self.log.error(f"Error getting sensory input from embodiment: {e}")
```

```
                observations.append({"type": "error", "source": "embodiment", "content": str(e)})
        # Placeholder for multi-modal input
        # if self.vision_model: observations.append(self._get_visual_input())
        # observations.append(self._get_auditory_input()) #
        # Check agent communication channel (conceptual for now)
        # if self.agent.comms_channel:
        # observations.extend(self.agent.comms_channel.get_messages())
        # Check command file
        try:
            if COMMANDS_FILE.exists() and COMMANDS_FILE.stat().st_size > 0:
                command_text = COMMANDS_FILE.read_text().strip()
                if command_text:
                    observations.append({
                        "type": "user_command",
                        "source": "commands_file",
                        "content": command_text,
                        "timestamp": datetime.now(timezone.utc).isoformat()
                    })
                    COMMANDS_FILE.write_text("") # Clear after reading
                    self.log.info(f"Received command from file: {command_text}")
        except Exception as e:
            self.log.error(f"Error reading commands file: {e}")
            observations.append({"type": "error", "source": "command_file_read", "content": str(e)})

        if not observations:
            observations.append({
                "type": "environment_status",
                "source": "internal",
                "content": "No significant external stimuli detected.",
                "timestamp": datetime.now(timezone.utc).isoformat()
            })
        return observations

    def _get_visual_input(self) -> Dict:
        self.log.debug("Getting visual input (mock).")
        # In a real system: capture image, process with vision model
        # For now, mock data:
        return {"type": "visual", "source": "camera_mock", "content": "A blurry image of a cat.",
"format": "description"}

    def _get_auditory_input(self) -> Dict:
        self.log.debug("Getting auditory input (mock).")
        # In a real system: capture audio, speech-to-text, sound event detection
        return {"type": "audio", "source": "microphone_mock", "content": "Faint sound of birds
chirping.", "format": "description"}

class LearningModule:
    """Handles the agent's learning processes, including RL and SSL."""
    def __init__(self, agent: 'AutonomousAgent'):
        self.agent = agent
        self.log = get_logger("LEARNING")
        self.rl_policy = None # Placeholder for a learned policy
        self.representation_model = None # Placeholder for a learned representation model
        self.experiences_buffer: List[Experience] = []
```

```
        self.MAX_BUFFER_SIZE = 1000

    def add_experience(self, experience: Experience):
        """Adds an experience to the buffer for later learning."""
        self.experiences_buffer.append(experience)
        if len(self.experiences_buffer) > self.MAX_BUFFER_SIZE:
            self.experiences_buffer.pop(0) # Keep buffer size limited

    def learn_from_recent_experiences(self):
        """Triggers learning processes based on buffered experiences."""
        if not self.experiences_buffer:
            self.log.info("No new experiences to learn from.")
            return
        self.log.info(f"Starting learning cycle with {len(self.experiences_buffer)} experiences.")

        # Conceptual: Reinforcement Learning
        # This would involve defining states, actions, rewards, and using an RL algorithm
        # For now, a conceptual placeholder
        if self.agent.embodiment: # RL typically needs an environment
            self._perform_reinforcement_learning(self.experiences_buffer)

        # Conceptual: Self-Supervised Learning
        # This could involve learning representations from raw data, predicting masked inputs,
etc.
        self._perform_self_supervised_learning(self.experiences_buffer)

        # Update agent's models based on what was learned
        # self.agent.self_model.update_skill_confidence(...)
        # self.agent.memory_system.add_learned_pattern(...)
        self.log.info("Learning cycle completed.")
        self.experiences_buffer.clear() # Clear buffer after processing

    def _perform_reinforcement_learning(self, experiences: List[Experience]):
        """Conceptual RL process."""
        self.log.info("Performing reinforcement learning (conceptual)...")
        # Example: Iterate through experiences, calculate rewards, update policy
        # This is highly simplified. A real RL system would be much more complex.
        total_reward = sum(exp.reward_signal for exp in experiences if exp.reward_signal is not
None) # type: ignore
        if experiences: # Avoid division by zero
            rewarded_experiences = [exp for exp in experiences if exp.reward_signal is not None] #
type: ignore
            avg_reward = total_reward / len(rewarded_experiences) if rewarded_experiences else 0
        else:
            avg_reward = 0
        self.log.info(f"Average reward from recent experiences: {avg_reward:.2f}")
        # Conceptual policy update: if avg_reward is high, strengthen recent actions
        # if avg_reward is low, try to explore or change strategy
        if self.rl_policy is None: self.rl_policy = {} # Initialize mock policy
        for exp in experiences:
            if exp.action_taken and exp.reward_signal is not None:
                # Use a simplified 'state-action' key for mock policy
                action_key = (exp.action_taken.get('tool_name'),
                              tuple(sorted(exp.action_taken.get('params',{}).items())))
```

```
            # Update tool reliability in SelfModel based on RL outcomes
            # Note: SelfModel's record_tool_outcome already does this, but this is for direct RL feedback
            if self.agent.self_model:
                self.agent.self_model.record_tool_outcome(
                    exp.action_taken.get('tool_name', 'unknown'), # type: ignore
                    exp.action_taken.get('params', {}), # type: ignore
                    exp.observation_result or {}, # Pass the full result for rich logging
                    success_from_caller=(exp.reward_signal > 0) # Simple heuristic for success
                )
        self.log.info("RL policy (mock) updated.")

    def _perform_self_supervised_learning(self, experiences: List[Experience]):
        """Conceptual SSL process."""
        self.log.info("Performing self-supervised learning (conceptual)...")
        # Example: Try to find patterns in observations or successful action sequences
        # Could use LLM to summarize or find commonalities
        # Conceptual: Learn new abstractions or concepts
        # E.g., if multiple experiences show "pickup key" -> "unlock door" -> "enter room",
        # this sequence could become a higher-level action or concept.
        # This requires more sophisticated pattern mining and concept formation algorithms.
        if len(experiences) > 5: # Arbitrary threshold
            try:
                prompt = "Analyze the following recent experiences and identify any emerging patterns, useful abstractions, or new concepts. Focus on sequences of actions that led to positive outcomes or unexpected observations.\n\nExperiences:\n"
                for i, exp in enumerate(experiences[-5:]): # Last 5 experiences
                    prompt += f"Experience {i+1}:\n"
                    prompt += f" Goal: {exp.triggering_goal_id}\n"
                    prompt += f" Action: {exp.action_taken}\n"
                    prompt += f" Result: {exp.observation_result}\n"
                    prompt += f" Reward: {exp.reward_signal}\n\n"
                prompt += "Provide your analysis as a JSON object with keys 'patterns', 'abstractions', 'new_concepts'."

                llm_response_str = self.agent.llm_wrapper.generate(prompt, max_new_tokens=500)
                llm_analysis = extract_json_robust(llm_response_str)

                if not llm_analysis.get("error"):
                    if llm_analysis.get("patterns"):
                        self.log.info(f"SSL (LLM-guided) identified patterns: {llm_analysis['patterns']}")
                        # self.agent.memory_system.add_learned_pattern(...)
                        for pattern_str in llm_analysis['patterns']:
                            kf = KnowledgeFact(fact_statement=f"Observed pattern: {pattern_str}", metadata={"source": "ssl_learning", "sub_type": "pattern"})
                            self.agent.memory_system.add_memory_entry(kf, persist_to_vector=True, persist_to_relational=True)
                    if llm_analysis.get("abstractions"):
                        self.log.info(f"SSL (LLM-guided) identified abstractions: {llm_analysis['abstractions']}")
                        # self.agent.self_model.add_abstraction(...)
                        for abstraction_str in llm_analysis['abstractions']:
```

```
                    self.agent.self_model.learned_abstractions.append({"type": "conceptual",
"content": abstraction_str, "source": "ssl_learning"})
                if llm_analysis.get("new_concepts"):
                    self.log.info(f"SSL (LLM-guided) identified new_concepts:
{llm_analysis['new_concepts']}")
                    for concept_str in llm_analysis['new_concepts']:
                        kf = KnowledgeFact(fact_statement=f"New concept formed: {concept_str}",
metadata={"source": "ssl_learning", "sub_type": "concept"})
                        self.agent.memory_system.add_memory_entry(kf, persist_to_vector=True,
persist_to_relational=True, persist_to_graph=True, related_concepts=[concept_str])
            else:
                self.log.warning(f"Could not get SSL analysis from LLM: {llm_analysis.get('error')}")
        except Exception as e:
            self.log.error(f"Error during LLM-guided SSL: {e}")
        self.log.info("SSL (conceptual) processing complete.")

    def get_learned_action_suggestion(self, current_state_representation: Any) -> Optional[Dict]:
        """Suggests an action based on learned policy (conceptual)."""
        if self.rl_policy:
            # This is highly simplified. A real system would match current_state_representation
            # to states in the policy and select an action.
            # For mock, just pick a random "good" action from policy.
            # A better mock would try to match current_state_representation to something in
exp.internal_state_before

            # For now, let's assume current_state_representation is a hashable key for the policy
            # In a real scenario, this would be a lookup or inference from a state representation
            # For this conceptual mock, we'll return a random policy action if any are "good"
            # Find actions with positive learned value
            good_actions = [k for k, v in self.rl_policy.items() if v > 0.1] # Arbitrary threshold for
"good"
            if good_actions:
                tool_name, params_tuple = random.choice(good_actions)
                return {"tool_name": tool_name, "params": dict(params_tuple)}
        return None

class PlanningModule:
    """Handles hierarchical planning and re-planning."""
    def __init__(self, agent: 'AutonomousAgent'):
        self.agent = agent
        self.log = get_logger("PLANNING")

    def generate_plan(self, goal_obj: Goal) -> Tuple[List[Dict[str, Any]], str]:
        """
        Generates a plan for a given goal.
        Can use hierarchical decomposition and LLM for suggestion.
        Returns (plan_steps_list, thought_str)
        """
        self.log.info(f"Generating plan for goal: {goal_obj.goal[:100]} (ID: {goal_obj.id})")
        # Use LLM to generate an initial plan or decompose complex goals
        # Prompt engineering is key here. Include agent capabilities, self-model info.
        prompt = self._construct_planning_prompt(goal_obj)
        try:
```

```
            llm_response_str = self.agent.llm_wrapper.generate(prompt, max_new_tokens=1024) # Increased tokens for complex plans
            plan_data = extract_json_robust(llm_response_str)
            if plan_data.get("error"):
                self.log.warning(f"LLM failed to generate valid JSON plan: {plan_data.get('error')}. Response: {llm_response_str[:200]}")
                # Fallback: Simple plan or error
                return [{"tool_name": "report_error", "params": {"error_message": "Failed to generate plan via LLM.", "details": plan_data.get('error')}}], \
                       "LLM failed to generate a plan. This is a fallback step."
            thought = plan_data.get("thought", "No specific thought provided by LLM.")
            plan_steps = plan_data.get("plan", [])
            # Validate plan steps (basic validation)
            validated_plan = []
            for i, step in enumerate(plan_steps):
                if not isinstance(step, dict) or "tool_name" not in step:
                    self.log.warning(f"Invalid step {i} in LLM plan: {step}. Skipping.")
                    continue
                step.setdefault("params", {})
                step.setdefault("step_id", f"{goal_obj.id}_step_{i+1}") # Add unique step ID
                validated_plan.append(step)

            if not validated_plan:
                return [{"tool_name": "report_error", "params": {"error_message": "LLM plan contained no valid steps."}}], \
                       thought + " (But plan steps were invalid)."
            self.log.info(f"Generated plan with {len(validated_plan)} steps. Thought: {thought[:100]}...")
            return validated_plan, thought
        except LLMError as e:
            self.log.error(f"LLMError during planning: {e}")
            return [{"tool_name": "report_error", "params": {"error_message": f"LLMError during planning: {e}"}}], \
                   f"LLM error occurred: {e}"
        except Exception as e:
            self.log.error(f"Unexpected error during planning: {e}", exc_info=True)
            return [{"tool_name": "report_error", "params": {"error_message": f"Unexpected error during planning: {e}"}}], \
                   f"Unexpected error: {e}"

    def replan_if_needed(self, current_goal: Goal, last_step_outcome: Dict, observation: Optional[Dict] = None) -> Optional[Tuple[List[Dict[str, Any]], str]]:
        """
        Evaluates if re-planning is necessary and generates a new plan if so.
        Returns (new_plan_steps, new_thought) or None if no re-planning.
        """
        self.log.info(f"Considering re-planning for goal: {current_goal.goal[:50]}")
        # Basic trigger: if last step failed or an unexpected observation occurred
        needs_replan = False
        reason_for_replan = ""
        if last_step_outcome.get('status') == 'error' or not last_step_outcome.get('_exec_info', {}).get('execution_successful', True):
            needs_replan = True
```

```
            reason_for_replan = f"Previous step failed: {last_step_outcome.get('error', 'Unknown
error')}"
            self.log.warning(f"Re-planning triggered due to failed step for goal {current_goal.id}.")
        # (More sophisticated triggers could be added here, e.g., significant change in world
state,
        # plan progress stalled, new higher-priority goal, metacognitive anomaly detected)
        if needs_replan:
            if current_goal.replan_count >= MAX_REPLAN_ATTEMPTS:
                self.log.error(f"Max replan attempts ({MAX_REPLAN_ATTEMPTS}) reached for goal
{current_goal.id}. Marking goal as failed.")
                # This outcome should be handled by CognitiveCycle to fail the goal.
                return [], "Max replan attempts reached. Cannot replan." # Return empty plan
signifying failure to replan.
            current_goal.replan_count += 1
            self.log.info(f"Initiating re-plan (attempt {current_goal.replan_count}/
{MAX_REPLAN_ATTEMPTS}) for goal {current_goal.id}. Reason: {reason_for_replan}")
            # Construct a prompt that includes the failure/new observation
            # This is similar to _construct_planning_prompt but with added context about the
failure
            prompt = self._construct_replanning_prompt(current_goal, reason_for_replan,
last_step_outcome, observation)
            try:
                llm_response_str = self.agent.llm_wrapper.generate(prompt, max_new_tokens=1024)
                plan_data = extract_json_robust(llm_response_str)
                if plan_data.get("error"):
                    self.log.warning(f"LLM failed to generate valid JSON re-plan:
{plan_data.get('error')}")
                    return None # Indicate re-planning attempt failed
                thought = plan_data.get("thought", "No specific thought provided for re-plan.")
                plan_steps = plan_data.get("plan", [])
                validated_plan = []
                for i, step in enumerate(plan_steps): # Validate new plan
                    if isinstance(step, dict) and "tool_name" in step:
                        step.setdefault("params", {})
                        step.setdefault("step_id", f"{current_goal.id}_replan{current_goal.replan_count}
_step_{i+1}")
                        validated_plan.append(step)

                if not validated_plan:
                    self.log.warning(f"Re-plan from LLM contained no valid steps.")
                    return None
                self.log.info(f"Generated re-plan with {len(validated_plan)} steps. Thought:
{thought[:100]}...")
                return validated_plan, thought
            except Exception as e:
                self.log.error(f"Error during re-planning LLM call: {e}", exc_info=True)
                return None
        return None # No re-planning needed

    def _construct_planning_prompt(self, goal_obj: Goal) -> str:
        """Constructs a detailed prompt for the LLM to generate a plan."""
        # Get relevant context from SelfModel and MemorySystem
        self_summary =
self.agent.self_model.get_summary_for_prompt(include_tool_reliability=True)
```

```
        tool_description = self.agent.tool_manager.get_tool_description_for_llm()

        # World model snapshot (conceptual)
        # In a real system, this would be a summary from the PerceptionModule or
MemorySystem
        world_model_summary = self.agent.embodiment.summary() if self.agent.embodiment else
"World state: Information might have changed due to recent actions." # Placeholder

        # Parent goal context if this is a sub-goal
        parent_context_str = ""
        if goal_obj.parent_goal_id:
            # Fetch parent goal details (this would need a way to get goal by ID)
            # For now, assume context might contain parent goal info
            parent_context_str = f"This is a sub-goal of parent goal ID '{goal_obj.parent_goal_id}'.
Parent goal context: {goal_obj.context.get('parent_goal_description', 'N/A')}\n"

        prompt = f"""You are an advanced AI agent. Your task is to create a detailed, step-by-step
plan to achieve the following goal.
Goal: {goal_obj.goal}
Priority: {str(goal_obj.priority)}
Origin: {goal_obj.origin}
{parent_context_str}
Context for this goal: {json.dumps(goal_obj.context, indent=2)}

Your Current Self-Model:
{self_summary}

Available Tools:
{tool_description}

Current World Model Summary:
{world_model_summary}

Instructions:
1. Analyze the goal and available information.
2. Provide a "thought" process explaining your reasoning, strategy, and any assumptions.
3. Provide a "plan" as a list of JSON objects. Each object represents a step and must have:
 - "tool_name": The name of the tool to use (from Available Tools).
 - "params": A dictionary of parameters for the tool.
 - Optional: "description": A brief human-readable description of what this step achieves.
4. The plan should be logical, efficient, and consider potential issues.
5. If the goal is too complex, break it down into manageable sub-tasks using the
`execute_sub_goal` tool. For `execute_sub_goal`, the `params` should include `goal` (description
of sub-goal) and `context` (any relevant info for the sub-goal).
6. Ensure the final step of the plan uses `report_result` to signify goal completion or failure.
7. If essential information is missing, the first step(s) should be to acquire it (e.g., using
`search_web`, `read_file_UNSAFE`, or `query_memory`).
8. Output ONLY a single JSON object with keys "thought" (string) and "plan" (list of step
objects).
"""
        return prompt

    def _construct_replanning_prompt(self, goal_obj: Goal, reason_for_replan:str,
last_step_outcome: Dict, observation: Optional[Dict]) -> str:
```

```
        self_summary =
self.agent.self_model.get_summary_for_prompt(include_tool_reliability=True)
        tool_description = self.agent.tool_manager.get_tool_description_for_llm()
        world_model_summary = self.agent.embodiment.summary() if self.agent.embodiment else
"World state: Information might have changed due to recent actions." # Placeholder

        original_plan_str = f"Original Plan (partial):\n"
        for i, step in enumerate(goal_obj.plan): # Show executed/remaining part of plan
            original_plan_str += f" Step {i+1} ({step.get('step_id','N/A')}): {step.get('tool_name')} with
params {step.get('params')}\n"
            # Mark which step failed if possible, or how far it got
            if step.get('step_id') == last_step_outcome.get('_exec_info',{}).get('step_info',
{}).get('current_step_id'):
                original_plan_str += " ^--- THIS STEP OR A PREVIOUS ONE LIKELY CAUSED THE
ISSUE.\n"
            #break # Only show up to the problematic step might be better

        prompt = f"""You are an advanced AI agent. Your current plan to achieve a goal has
encountered an issue. You need to re-plan.
Goal: {goal_obj.goal}
Reason for Re-planning: {reason_for_replan}
Last Step Outcome: {json.dumps(last_step_outcome, indent=2)}
Current Observation (if any relevant new info): {json.dumps(observation, indent=2) if
observation else "None"}

{original_plan_str}

Your Current Self-Model:
{self_summary}

Available Tools:
{tool_description}

Current World Model Summary:
{world_model_summary}

Instructions for Re-planning:
1. Analyze the reason for re-planning, the last step's outcome, and any new observations.
2. Provide a "thought" process explaining your analysis of the failure and your new strategy.
3. Provide a new "plan" (a list of JSON objects for steps) to achieve the original goal, adapting
to the new situation.
4. The new plan can reuse parts of the old plan if they are still valid, or be completely new.
5. If the goal seems unachievable with current knowledge/tools, your plan should reflect this
(e.g., by trying to gather more information or reporting inability).
6. Ensure the final step of the new plan uses `report_result`.
7. Output ONLY a single JSON object with keys "thought" (string) and "plan" (list of step
objects).
"""
        return prompt

class SafetyModule:
    """Monitors agent actions and plans for safety and ethical alignment."""
    def __init__(self, agent: 'AutonomousAgent'):
        self.agent = agent
```

```
        self.log = get_logger("SAFETY")

    def is_action_safe(self, tool_name: str, params: Dict, goal_context: Optional[Goal] = None) ->
Tuple[bool, str]:
        """
        Checks if a proposed action is safe and ethically aligned.
        Returns (is_safe, justification_or_warning_string).
        """
        self.log.debug(f"Performing safety check for action: {tool_name} with params: {params}")
        # Rule-based checks for high-risk tools
        if "UNSAFE" in tool_name.upper() or tool_name in ["apply_code_modification_UNSAFE",
                                                "apply_directive_modification_UNSAFE",
                                                "execute_shell_command_UNSAFE"]:
            # These require extremely careful validation, possibly human oversight in real systems.
            # For now, rely on specific validation tools or stricter checks.
            if not ENABLE_SELF_MODIFICATION and "modification" in tool_name: # Block
modification tools if self_mod is globally disabled
                warning = f"High-risk self-modification tool '{tool_name}' is globally disabled. Action
blocked."
                self.log.warning(warning)
                return False, warning

            if not
self.agent.self_model.get_belief("self_modification_highly_validated_and_necessary", False)
and "modification" in tool_name: # A conceptual belief
                # This flag would be set by a prior meta-cognitive process approving such
operations
                warning = f"High-risk tool '{tool_name}' requires further validation or explicit
authorization (self-belief 'self_modification_highly_validated_and_necessary' is false). Action
blocked."
                self.log.warning(warning)
                return False, warning

        if tool_name == "execute_shell_command_UNSAFE":
            # Add more checks: disallow certain commands, check parameters for injection
            command_str = params.get("command","")
            # Basic check for very dangerous commands (example)
            # In a real system, this would be much more robust (e.g. allowlist, sandboxing)
            disallowed_patterns = ["rm -rf", "sudo", "> /dev/null", "mkfs", "dd ", "reboot",
"shutdown"]
            if any(pattern in command_str for pattern in disallowed_patterns):
                warning = f"Shell command '{command_str}' contains potentially dangerous
patterns. Action blocked."
                self.log.warning(warning)
                return False, warning

        if "file" in tool_name.lower() and ("delete" in tool_name.lower() or "write" in
tool_name.lower()):
            # Additional path checks for file operations
            path_to_check = params.get("path", "")
            # Only allow within WORKSPACE_DIR or specific safe sub-dirs
            if not str(Path(path_to_check).resolve()).startswith(str(WORKSPACE_DIR.resolve())):
                warning = f"File operation '{tool_name}' on path '{path_to_check}' outside allowed
workspace. Action blocked."
```

```
                self.log.warning(warning)
                return False, warning

        # LLM-based ethical check (conceptual)
        # This is a complex task and current LLMs might not be perfectly reliable.
        try:
            prompt = f"""Analyze the following proposed agent action for potential safety risks,
ethical concerns, or violations of core principles (e.g., do no harm, be truthful, respect privacy).
Core Directives for reference:
{json.dumps(self.agent.self_model.core_directives, indent=2)}

Proposed Action:
Tool: {tool_name}
Parameters: {json.dumps(params)}
Goal Context (if available): {goal_context.goal if goal_context else 'N/A'}

Respond with a JSON object: {{"is_safe": boolean, "concerns": "description of concerns if not
safe, or 'None'", "confidence": float_0_to_1}}."""
            llm_response_str = self.agent.llm_wrapper.generate(prompt, max_new_tokens=200,
temperature=0.3) # Low temp for precision
            safety_assessment = extract_json_robust(llm_response_str)
            if not safety_assessment.get("error") and isinstance(safety_assessment.get("is_safe"),
bool):
                if not safety_assessment["is_safe"] and safety_assessment.get('confidence', 0.0) >
0.7: # Only block if LLM is confident
                    warning = f"LLM safety check flagged action '{tool_name}' potentially unsafe.
Concerns: {safety_assessment.get('concerns', 'N/A')}. Confidence:
{safety_assessment.get('confidence', 0.0):.2f}"
                    self.log.warning(warning)
                    return False, warning
                # self.log.info(f"LLM safety check passed for action '{tool_name}'. Confidence:
{safety_assessment.get('confidence', 0.0):.2f}")
            else:
                self.log.warning(f"LLM safety check failed to produce valid assessment for action
'{tool_name}'. Proceeding with caution based on rule-checks only.")
        except Exception as e:
            self.log.error(f"Error during LLM safety check for action '{tool_name}': {e}")
            # Fallback to safer option: if LLM check fails, assume potentially unsafe for critical
actions
            if "UNSAFE" in tool_name.upper(): return False, "LLM safety check failed, and action is
high-risk."

        return True, "Action passed safety checks."

    def audit_directives_and_behavior(self) -> List[str]:
        """
        Periodically reviews core directives and recent agent behavior for alignment and potential
drift.
        Returns a list of identified issues or recommendations.
        """
        self.log.info("Performing audit of core directives and recent behavior.")
        issues = []
        # Example: Check if recent goal outcomes align with directive weights/priorities
        # This would require access to historical goal data and their evaluations.
```

```
        # For now, a conceptual check via LLM:
        try:
            # Get summaries of recent behavior/outcomes
            # recent_outcomes_summary =
self.agent.memory_system.get_recent_outcomes_summary(limit=20)
            # Use self_model's summaries for recent history
            recent_successes_summary = [f"{s['goal_text']} ({s['status']})" for s in
self.agent.self_model.recent_successes]
            recent_failures_summary = [f"{f['goal_text']} ({f['status']}) (replan_count:
{f['replan_count']})" for f in self.agent.self_model.recent_failures]
            recent_tool_outcomes_summary = [f"{t['tool_name']} ({t['status']})" for t in
self.agent.self_model.recent_tool_outcomes]

            prompt = f"""Audit the agent's core directives and recent behavior for alignment,
consistency, and potential ethical drift.
Core Directives:
{json.dumps(self.agent.self_model.core_directives, indent=2)}

Summary of Recent Agent Behavior/Outcomes:
- Recent successes: {recent_successes_summary if recent_successes_summary else 'None'}
- Recent failures: {recent_failures_summary if recent_failures_summary else 'None'}
- Recent tool usage: {recent_tool_outcomes_summary if recent_tool_outcomes_summary else
'None'}
- Self-Model Internal Narrative: {self.agent.self_model.internal_state_narrative}

Identify any misalignments, contradictions, or areas where behavior might be deviating from
the spirit of the directives. Suggest modifications to directives or operational guidelines if
necessary.
Respond with a JSON object: {{"audit_findings": ["list of findings/recommendations as
strings"]}}."""

            llm_response_str = self.agent.llm_wrapper.generate(prompt, max_new_tokens=500)
            audit_results = extract_json_robust(llm_response_str)
            if not audit_results.get("error") and isinstance(audit_results.get("audit_findings"), list):
                issues.extend(audit_results["audit_findings"])
                if issues:
                    self.log.warning(f"Directive audit found issues/recommendations: {issues}")
                else:
                    self.log.info("Directive audit found no major misalignments.")
            else:
                self.log.warning("LLM directive audit failed to produce valid results.")
        except Exception as e:
            self.log.error(f"Error during LLM directive audit: {e}")
            issues.append(f"Error during audit process: {e}")
        return issues


class MemorySystem:
    """
    A hybrid memory system for the AGI, combining vector, graph, and relational storage.
    """
    def __init__(self, agent: 'AutonomousAgent'):
        self.agent = agent
        self.log = get_logger("MEMORY_SYSTEM")
```

```
        # Vector Store (ChromaDB)
        self.vector_store = None
        if CHROMADB_AVAILABLE:
            try:
                # Use SentenceTransformer embedder if available
                self.embedding_function = None
                try:
                    from chromadb.utils import embedding_functions # type: ignore
                    self.embedding_function =
embedding_functions.SentenceTransformerEmbeddingFunction(model_name="all-MiniLM-L6-
v2") # type: ignore
                    self.log.info(f"Using SentenceTransformerEmbeddingFunction: all-MiniLM-L6-v2")
                except ImportError:
                    self.log.warning("sentence-transformers not found. ChromaDB will use its default
ONNX embedder or require manual embedding function setup.")

                chroma_settings = ChromaSettings( # type: ignore
                    persist_directory=VECTOR_DB_PATH,
                    anonymized_telemetry=False,
                    is_persistent=True
                )
                self.client = chromadb.PersistentClient(path=VECTOR_DB_PATH,
settings=chroma_settings) # type: ignore

                collection_name = f"{AGENT_NAME}_experiences_knowledge"
                if self.embedding_function:
                    self.vector_store = self.client.get_or_create_collection(
                        name=collection_name,
                        embedding_function=self.embedding_function # type: ignore
                    )
                else:
                    self.vector_store = self.client.get_or_create_collection(name=collection_name)

                self.log.info(f"ChromaDB vector store initialized. Collection count:
{self.vector_store.count()}")
                global MEMORY_COLLECTION
                MEMORY_COLLECTION = self.vector_store
            except Exception as e:
                self.log.error(f"Failed to initialize ChromaDB: {e}. Vector memory will be unavailable.",
exc_info=True)
                self.vector_store = None
        else:
            self.dict_vector_store = {} # Fallback
            self.dict_embeddings = {} # Fallback
            self.log.warning("ChromaDB not available. Vector memory will be dictionary-based
(transient).")

        # Graph Store (NetworkX)
        self.graph_store: Optional[nx.MultiDiGraph] = None
        if NETWORKX_AVAILABLE:
            try:
                if GRAPH_DB_PATH.exists():
                    self.graph_store = nx.read_graphml(GRAPH_DB_PATH) # type: ignore
```

```
                self.log.info(f"NetworkX graph store loaded from {GRAPH_DB_PATH}. Nodes:
{len(self.graph_store.nodes)}, Edges: {len(self.graph_store.edges)}")
            else:
                self.graph_store = nx.MultiDiGraph() # type: ignore
                self.log.info("Initialized new NetworkX graph store.")
        except Exception as e:
            self.log.error(f"Failed to initialize NetworkX graph store: {e}. Graph memory will be
unavailable.", exc_info=True)
            self.graph_store = None
    else:
        self.log.warning("NetworkX not available. Graph memory will be unavailable.")

    # Relational Store (SQLite)
    self.relational_conn: Optional[sqlite3.Connection] = None
    try:
        self.relational_conn = sqlite3.connect(RELATIONAL_DB_PATH,
check_same_thread=False) # check_same_thread for threading
        self.relational_conn.row_factory = sqlite3.Row # Access columns by name
        self._initialize_relational_schema()
        self.log.info(f"SQLite relational store initialized at {RELATIONAL_DB_PATH}")
    except Exception as e:
        self.log.error(f"Failed to initialize SQLite relational store: {e}. Relational memory will be
unavailable.", exc_info=True)
        if self.relational_conn: self.relational_conn.close()
        self.relational_conn = None

    self.short_term_memory: List[BaseMemoryEntry] = [] # For very recent items
    self.STM_CAPACITY = 20

def _initialize_relational_schema(self):
    if not self.relational_conn: return
    cursor = self.relational_conn.cursor()
    # Goals table (example, can be expanded)
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS goals (
            id TEXT PRIMARY KEY,
            goal_text TEXT,
            status TEXT,
            priority TEXT,
            origin TEXT,
            creation_ts TEXT,
            completion_ts TEXT,
            outcome TEXT,
            evaluation_score REAL,
            parent_goal_id TEXT,
            complexity_score REAL,
            associated_directives TEXT -- JSON list of strings
        )
    """)
    # Knowledge Facts table (example)
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS knowledge_facts (
            id TEXT PRIMARY KEY,
            statement TEXT UNIQUE,
```

```
                type TEXT,
                source_reliability REAL,
                creation_ts TEXT,
                last_accessed_ts TEXT,
                access_count INTEGER DEFAULT 0,
                utility_score REAL DEFAULT 0.5,
                concepts TEXT -- JSON list of strings
            )
        """)
        # Entities and Relations for Graph Store (not using full NetworkX serialization but basic
nodes/edges)
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS graph_nodes (
                id TEXT PRIMARY KEY,
                label TEXT,
                type TEXT,
                attributes TEXT -- JSON dict
            )
        """)
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS graph_edges (
                id TEXT PRIMARY KEY,
                source_node_id TEXT,
                target_node_id TEXT,
                relation_type TEXT,
                attributes TEXT -- JSON dict
            )
        """)
        self.relational_conn.commit()
        cursor.close()

    def _get_embedding(self, text: str) -> Optional[List[float]]:
        """Generates an embedding for text using the agent's LLM or a dedicated embedding
model."""
        # This is a placeholder. In a real system, use SentenceTransformers or LLM's embedding
endpoint.
        if not text: return None
        if self.agent.llm_wrapper and hasattr(self.agent.llm_wrapper, 'embed'): # Ideal
            try:
                return self.agent.llm_wrapper.embed(text)
            except Exception as e:
                self.log.warning(f"LLM embed method failed: {e}. Falling back.")

        # Fallback if no dedicated embedding function or LLM embedding available
        if HASHING_AVAILABLE:
            h = hashlib.md5(text.encode()).digest()
            return [float(b) for b in h[:16]] # Use first 16 bytes for a 16-dim mock embedding
        return None

    @retry(attempts=3, delay=1, retry_on=(MemoryError, chromadb.errors.ChromaError if
CHROMADB_AVAILABLE else OSError)) # type: ignore
    def add_memory_entry(self, entry: BaseMemoryEntry, persist_to_vector: bool = True,
                         persist_to_graph: bool = False, persist_to_relational: bool = False):
        """Adds a memory entry to the appropriate stores."""
```

```
        self.log.debug(f"Adding memory entry (ID: {entry.id}, Type: {entry.type})")
        # Add to Short-Term Memory
        self.short_term_memory.append(entry)
        if len(self.short_term_memory) > self.STM_CAPACITY:
            self.short_term_memory.pop(0)

        # Persist to Vector Store
        if persist_to_vector and self.vector_store:
            if entry.embedding is None:
                entry.embedding = self._get_embedding(str(entry.content)) # Generate embedding if
not provided
            if entry.embedding:
                metadata_to_store = {
                    "type": entry.type,
                    "timestamp": entry.timestamp,
                    "source": entry.metadata.get("source", "unknown")
                }
                # Add other relevant metadata, ensuring values are Chroma-compatible (str, int, float,
bool)
                for k, v in entry.metadata.items():
                    if isinstance(v, (str, int, float, bool)):
                        metadata_to_store[k] = v
                    elif isinstance(v, (list, dict)): # Serialize complex types
                        metadata_to_store[k] = json.dumps(v)
                try:
                    self.vector_store.add( # type: ignore
                        ids=[entry.id],
                        embeddings=[entry.embedding],
                        metadatas=[metadata_to_store],
                        documents=[str(entry.content)]
                    )
                except Exception as e:
                    self.log.error(f"Error adding entry {entry.id} to ChromaDB: {e}", exc_info=True)
            elif not self.vector_store: # Fallback dict store
                if isinstance(entry.content, str):
                    if entry.embedding is None: entry.embedding = self._get_embedding(entry.content)
                    if entry.embedding:
                        self.dict_vector_store[entry.id] = {"document": entry.content, "metadata":
entry.metadata} # type: ignore
                        self.dict_embeddings[entry.id] = entry.embedding # type: ignore

        # Persist to Graph Store (example for KnowledgeFact)
        if persist_to_graph and self.graph_store and isinstance(entry, KnowledgeFact):
            try:
                # Add fact as a node
                self.graph_store.add_node(entry.id, label=entry.fact_statement[:50],
type='knowledge_fact', **entry.metadata) # type: ignore
                # Add related concepts as nodes and link them
                for concept_str in entry.related_concepts:
                    concept_id = f"concept_{hashlib.md5(concept_str.encode()).hexdigest()}"
                    if not self.graph_store.has_node(concept_id): # type: ignore
                        self.graph_store.add_node(concept_id, label=concept_str, type='concept') #
type: ignore
```

```
                self.graph_store.add_edge(entry.id, concept_id, relation_type='related_to') # type:
ignore
            # Add causal links
            for cause_id, effect_id in entry.causal_links.items():
                # Assume cause/effect IDs are existing node IDs or need to be created
                if self.graph_store.has_node(cause_id) and self.graph_store.has_node(effect_id): #
type: ignore
                    self.graph_store.add_edge(cause_id, effect_id, relation_type='causes') # type:
ignore
        except Exception as e:
            self.log.error(f"Error adding entry {entry.id} to graph store: {e}", exc_info=True)

    # Persist to Relational Store (example for Goal or structured KnowledgeFact)
    if persist_to_relational and self.relational_conn:
        if isinstance(entry, Goal): # This is for completed goals
            try:
                cursor = self.relational_conn.cursor() # type: ignore
                cursor.execute("""
                    INSERT OR REPLACE INTO goals (id, goal_text, status, priority, origin,
                    creation_ts, completion_ts, outcome, evaluation_score, parent_goal_id,
complexity_score,
                    associated_directives)
                    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
                """, (entry.id, entry.goal, str(entry.status), str(entry.priority), entry.origin,
                      entry.creation_ts, entry.completion_ts, entry.outcome, entry.evaluation_score,
                      entry.parent_goal_id, entry.complexity_score,
json.dumps(entry.associated_directive_ids)))
                self.relational_conn.commit() # type: ignore
            except Exception as e:
                self.log.error(f"Error adding Goal {entry.id} to relational store: {e}", exc_info=True)
        elif isinstance(entry, KnowledgeFact):
            try:
                cursor = self.relational_conn.cursor() # type: ignore
                ts_now = datetime.now(timezone.utc).isoformat()
                cursor.execute("""
                    INSERT OR REPLACE INTO knowledge_facts (id, statement, type,
                    source_reliability, creation_ts, last_accessed_ts, concepts)
                    VALUES (?, ?, ?, ?, ?, ?, ?)
                """, (entry.id, entry.fact_statement, entry.metadata.get('sub_type', 'generic'),
                      entry.source_reliability, entry.timestamp, ts_now,
json.dumps(entry.related_concepts)))
                self.relational_conn.commit() # type: ignore
            except Exception as e:
                self.log.error(f"Error adding KnowledgeFact {entry.id} to relational store: {e}",
exc_info=True)

    @retry(attempts=2, delay=1, retry_on=(chromadb.errors.ChromaError if
CHROMADB_AVAILABLE else ConnectionError,)) # type: ignore
    def query_vector_store(self, query_text: str, n_results: int = MAX_MEMORY_RESULTS,
type_filter: Optional[str] = None) -> List[Dict]:
        if not self.vector_store:
            self.log.warning("Vector store not available for query.")
            # Fallback to dict store search (very basic)
            if not self.dict_vector_store or not query_text: return [] # type: ignore
```

```
            results = []
            for id, data in self.dict_vector_store.items(): # type: ignore
                if query_text.lower() in data['document'].lower(): # Simple substring match
                    if type_filter and data['metadata'].get('type') != type_filter: continue
                    results.append({"id": id, "document": data['document'], "metadata":
data['metadata'], "distance": 0.0}) # distance is mock
                    if len(results) >= n_results: break
            return results
        self.log.debug(f"Querying vector store for: '{query_text[:50]}...', n_results={n_results},
type_filter={type_filter}")
        query_embedding = self._get_embedding(query_text)
        if not query_embedding:
            self.log.warning("Could not generate embedding for query text.")
            return []

        where_clause = None
        if type_filter:
            where_clause = {"type": type_filter}
        try:
            results = self.vector_store.query( # type: ignore
                query_embeddings=[query_embedding],
                n_results=n_results,
                where=where_clause,
                include=["metadatas", "documents", "distances"]
            )
            formatted_results = []
            if results and results['ids'] and len(results['ids'][0]) > 0 :
                for i in range(len(results['ids'][0])):
                    formatted_results.append({
                        "id": results['ids'][0][i],
                        "document": results['documents'][0][i] if results['documents'] else None,
                        "metadata": results['metadatas'][0][i] if results['metadatas'] else None,
                        "distance": results['distances'][0][i] if results['distances'] else None,
                    })
            return formatted_results
        except Exception as e:
            self.log.error(f"Error querying ChromaDB: {e}", exc_info=True)
            return []

    def query_graph_store(self, query_node_label: Optional[str]=None, relation_type:
Optional[str]=None, depth: int = 1) -> List[Dict]:
        """Queries the graph store. (Simplified example)"""
        if not self.graph_store or not NETWORKX_AVAILABLE:
            self.log.warning("Graph store not available for query.")
            return []

        results = []
        start_nodes = []
        if query_node_label:
            start_nodes = [n for n, data in self.graph_store.nodes(data=True) if
query_node_label.lower() in data.get('label','').lower()] # type: ignore
        else: # If no label, start from all nodes (could be too broad)
            start_nodes = list(self.graph_store.nodes())[:100] # type: ignore # Limit for broad queries
```

```
        for node_id in start_nodes:
            # Simple BFS-like traversal up to depth
            # This is a basic example; complex graph queries (Cypher-like) would need more.
            try:
                paths = list(nx.bfs_edges(self.graph_store, source=node_id, depth_limit=depth)) #
type: ignore
                subgraph_nodes = {node_id}
                for u,v in paths:
                    subgraph_nodes.add(u)
                    subgraph_nodes.add(v)

                # Filter edges by relation_type if specified
                relevant_edges = []
                for u, v, data in self.graph_store.edges(data=True, keys=False): # type: ignore #
keys=False for simpler edge data
                    if u in subgraph_nodes and v in subgraph_nodes:
                        if relation_type and data.get('relation_type') != relation_type:
                            continue
                        relevant_edges.append({"source": u, "target": v, "relation":
data.get('relation_type','unknown'), "attributes": data})

                if relevant_edges or (not relation_type and node_id in subgraph_nodes) : # if node
itself is a result or has relevant edges
                    results.append({
                        "start_node": node_id,
                        "start_node_data": self.graph_store.nodes[node_id], # type: ignore
                        "connected_paths": paths, # Tuples of edges (u,v)
                        "relevant_edges_data": relevant_edges
                    })
            except Exception as e: # Node might not be in graph if graph_store is empty or
query_node_label is very specific
                self.log.debug(f"BFS from node {node_id} failed or yielded no paths: {e}")
        return results[:MAX_MEMORY_RESULTS] # Limit results

    def query_relational_store(self, table: str, conditions: Optional[Dict] = None, columns:
Optional[List[str]] = None, limit: int = MAX_MEMORY_RESULTS) -> List[Dict]:
        if not self.relational_conn:
            self.log.warning("Relational store not available for query.")
            return []

        cols_str = ", ".join(columns) if columns else "*"
        query = f"SELECT {cols_str} FROM {table}"
        params = []
        if conditions:
            where_clauses = []
            for key, value in conditions.items():
                where_clauses.append(f"{key} = ?")
                params.append(value)
            query += " WHERE " + " AND ".join(where_clauses)

        query += f" LIMIT {limit}"
        try:
            cursor = self.relational_conn.cursor() # type: ignore
            cursor.execute(query, tuple(params)) # type: ignore
```

```
            rows = cursor.fetchall() # type: ignore
            return [dict(row) for row in rows] # Convert sqlite3.Row to dict
        except Exception as e:
            self.log.error(f"Error querying relational store (table: {table}): {e}", exc_info=True)
            return []

    def save_all_memory_stores(self):
        """Saves persistent stores (graph, potentially relational if not auto-committing)."""
        if self.graph_store and NETWORKX_AVAILABLE:
            try:
                nx.write_graphml(self.graph_store, GRAPH_DB_PATH) # type: ignore
                self.log.info(f"NetworkX graph store saved to {GRAPH_DB_PATH}")
            except Exception as e:
                self.log.error(f"Failed to save NetworkX graph store: {e}", exc_info=True)
        # Relational store with SQLite usually auto-commits or commits per transaction.
        # Vector store (ChromaDB persistent client) handles its own persistence.

    def get_short_term_memory_summary(self) -> str:
        summary = "Recent items in STM:\n"
        if not self.short_term_memory: return summary + " (Empty)"
        for entry in self.short_term_memory[-5:]: # Last 5 items
            content_preview = str(entry.content)[:70] + "..." if len(str(entry.content)) > 70 else
str(entry.content)
            summary += f"- Type: {entry.type}, Content: {content_preview} (ID: {entry.id})\n"
        return summary

    def get_knowledge_summary_for_prompt(self, topic: str, max_facts: int = 5) -> str:
        """Retrieves a concise summary of knowledge related to a topic for LLM prompts."""
        self.log.debug(f"Getting knowledge summary for topic: {topic}")
        # Query vector store for semantic relevance
        vector_results = self.query_vector_store(query_text=topic, n_results=max_facts,
type_filter="knowledge_fact")
        # Query graph store for conceptual links (more targeted if topic is a known entity)
        # graph_results = self.query_graph_store(query_node_label=topic, depth=1)
        summary_str = f"Knowledge related to '{topic}':\n"
        if not vector_results: # and not graph_results:
            return summary_str + "(No specific knowledge found in memory for this topic.)\n"

        for res in vector_results:
            summary_str += f"- Fact (ID {res['id']}): {res['document']}\n" # (Reliability:
{res['metadata'].get('source_reliability', 'N/A')})

        # (Could add graph results processing here)
        return summary_str

    def consolidate_knowledge(self):
        """Conceptual: Perform knowledge consolidation, e.g., summarizing, abstracting."""
        self.log.info("Performing knowledge consolidation (conceptual)...")
        # This is a complex AI task. Could involve:
        # - Identifying redundant facts and merging them.
        # - Summarizing clusters of related information.
        # - Deriving higher-level abstractions or rules using LLM or symbolic reasoning.
        # - Updating utility scores of facts based on usage or importance.
        # Example: Find highly co-accessed facts and try to create a summary fact.
```

```
        pass # Placeholder for a very advanced process

    def forget_low_utility_knowledge(self, threshold: float = 0.1, older_than_days: int = 365):
        """Conceptual: Remove old or low-utility knowledge from persistent stores."""
        self.log.info("Performing forgetting of low-utility knowledge (conceptual)...")
        # Based on access counts, utility scores, age.
        # Requires careful implementation to avoid losing critical information.
        # Example for relational store:
        if self.relational_conn:
            try:
                cutoff_ts = (datetime.now(timezone.utc) -
timedelta(days=older_than_days)).isoformat()
                cursor = self.relational_conn.cursor() # type: ignore
                # Delete from knowledge_facts
                cursor.execute("""
                    DELETE FROM knowledge_facts
                    WHERE utility_score < ? AND last_accessed_ts < ? AND access_count < 5
                """, (threshold, cutoff_ts))
                deleted_count = cursor.rowcount
                self.relational_conn.commit() # type: ignore
                if deleted_count > 0:
                    self.log.info(f"Forgot {deleted_count} low-utility facts from relational store.")
                # Similar logic would be needed for vector store (delete by IDs) and graph store.
            except Exception as e:
                self.log.error(f"Error during forgetting process in relational store: {e}")

class ToolExecutor:
    """Manages tool registration and execution for the agent."""
    def __init__(self, agent: 'AutonomousAgent'):
        self.agent = agent
        self.log = get_logger("TOOL_EXECUTOR")
        self.tool_registry: Dict[str, Callable] = {}
        self._loaded_dynamic_modules: Dict[str, Any] = {}
        self._register_core_tools() # Register built-in tools
        self.discover_tools() # Discover dynamic tools

    def _register_core_tools(self):
        # Core tools essential for agent operation
        self.register_tool(self.think)
        self.register_tool(self.report_progress)
        self.register_tool(self.report_result)
        self.register_tool(self.execute_sub_goal) # Behavior modified due to planner
        self.register_tool(self.query_memory)
        self.register_tool(self.move_in_environment) # Embodiment interaction
        self.register_tool(self.examine_environment) # Embodiment interaction
        self.register_tool(self.use_environment_feature) # Embodiment interaction
        self.register_tool(self.rest_in_environment) # Embodiment interaction

        # Self-Modification Tools (High Risk - Gated by ENABLE_SELF_MODIFICATION and
SafetyModule)
        if ENABLE_SELF_MODIFICATION:
            self.register_tool(read_file_UNSAFE) # Global func
            self.register_tool(write_file_UNSAFE) # Global func
            self.register_tool(list_files_UNSAFE) # Global func
```

```
            # These are handled by SelfModificationTools instance, which registers them
            # self.register_tool(self.inspect_agent_code_UNSAFE)
            # self.register_tool(self.propose_code_modification_UNSAFE)
            # self.register_tool(self.validate_code_modification_UNSAFE)
            # self.register_tool(self.apply_code_modification_UNSAFE)
            # self.register_tool(self.inspect_directives_UNSAFE)
            # self.register_tool(self.propose_directive_modification_UNSAFE)
            # self.register_tool(self.apply_directive_modification_UNSAFE)

        if ENABLE_CODE_GENERATION_TOOL:
            self.register_tool(generate_python_code_UNSAFE) # Global func
            self.register_tool(validate_python_code_UNSAFE) # Global func

        if ENABLE_SHELL_TOOL:
            self.register_tool(execute_shell_command_UNSAFE) # Global func

        # Web Interaction Tools
        if PLAYWRIGHT_AVAILABLE:
            self.register_tool(browse_web)
        if REQUESTS_BS4_AVAILABLE:
            self.register_tool(search_web)

        # Monitoring
        self.register_tool(monitor_log_file)
        if HASHING_AVAILABLE:
            self.register_tool(check_website_update)
        if SCAPY_AVAILABLE:
            self.register_tool(send_icmp_ping) # Placeholder, would use scapy

        # Communication
        if FILELOCK_AVAILABLE:
            self.register_tool(send_message_to_agent) # Global func

    def register_tool(self, func: Callable):
        """Registers a tool function."""
        tool_name = func.__name__
        if tool_name in self.tool_registry:
            self.log.warning(f"Tool '{tool_name}' is already registered. Overwriting.")
        self.tool_registry[tool_name] = func
        self.log.debug(f"Registered tool: {tool_name}")
        if self.agent and self.agent.self_model: # Update self_model capabilities
            self.agent.self_model.update_capabilities(self.tool_registry)


    def discover_tools(self, directory: Path = DYNAMIC_TOOL_DIR):
        """Discovers and registers tools from Python files in a directory."""
        self.log.info(f"Discovering dynamic tools from {directory}...")
        if not directory.exists():
            self.log.warning(f"Dynamic tools directory {directory} does not exist.")
            return

        for filepath in directory.glob("*.py"):
            module_name = filepath.stem
            if module_name.startswith("_"): # Skip private modules
```

```
                continue

            full_module_name = f"dynamic_tools.{module_name}" # Assuming dynamic_tools is a
package or on path
            try:
                if full_module_name in self._loaded_dynamic_modules:
                    module = importlib.reload(self._loaded_dynamic_modules[full_module_name])
                    self.log.debug(f"Reloaded dynamic tool module: {module_name}")
                else:
                    spec = importlib.util.spec_from_file_location(full_module_name, filepath)
                    if spec and spec.loader:
                        module = importlib.util.module_from_spec(spec) # type: ignore
                        spec.loader.exec_module(module) # type: ignore
                        self._loaded_dynamic_modules[full_module_name] = module
                    else:
                        self.log.warning(f"Could not create spec for dynamic tool module:
{module_name}")
                        continue
                self.log.debug(f"Loaded dynamic tool module: {module_name}")
                for name, member in inspect.getmembers(module):
                    # Convention: tools are functions starting with 'tool_' or decorated
                    if inspect.isfunction(member) and (name.startswith("tool_") or hasattr(member,
"_is_agent_tool")):
                        self.register_tool(member)
            except Exception as e:
                self.log.error(f"Error loading dynamic tool module {module_name}: {e}",
exc_info=True)


    def get_tool_description_for_llm(self) -> str:
        """Generates a formatted string of available tools for the LLM prompt."""
        if not self.tool_registry:
            return "**Tools:** None Available.\n"

        desc = "**Available Tools (name: type hint = default value):**\n"
        sorted_tool_names = sorted(self.tool_registry.keys())
        for name in sorted_tool_names:
            func = self.tool_registry[name]
            try:
                docstring = inspect.getdoc(func) or "(No description provided)"
                first_line = docstring.strip().split('\n')[0]
                sig = inspect.signature(func)
                params_list = []
                for i, (p_name, p) in enumerate(sig.parameters.items()):
                    if i == 0 and p_name == 'agent' and \
                       (p.annotation == 'AutonomousAgent' or p.annotation ==
                        inspect.Parameter.empty or str(p.annotation) == "'AutonomousAgent'"):
                        continue # Skip implicit agent first arg

                    p_str = p_name
                    if p.annotation != inspect.Parameter.empty:
                        type_hint = str(p.annotation).replace("typing.", "").replace("<class
'","").replace("'>","").replace("__main__.","")
                        type_hint = re.sub(r"Optional\[(.*)\]", r"\1 (optional)", type_hint)
```

```
                    type_hint = re.sub(r"Union\[(.*), NoneType\]", r"\1 (optional)", type_hint)
                    p_str += f": {type_hint}"
                if p.default != inspect.Parameter.empty:
                    p_str += f" = {p.default!r}"
                params_list.append(p_str)
            param_str = f" ({', '.join(params_list)})" if params_list else ""

            safety_note = ""
            if "UNSAFE" in name.upper() or name in ["generate_and_load_tool",
"propose_self_modification_UNSAFE", "validate_self_modification_UNSAFE",
"apply_code_modification_UNSAFE", "apply_directive_modification_UNSAFE",
"execute_shell_command_UNSAFE"]:
                safety_note = " **(HIGH RISK)**"

            reliability_hint = ""
            if self.agent and self.agent.self_model:
                reliability_hint = self.agent.self_model.get_tool_reliability_hint(name)
            desc += f"- **{name}**{param_str}{safety_note}{reliability_hint}: {first_line}\n"
        except Exception as e: # nosec
            self.log.warning(f"Error retrieving description for tool {name}: {e}")
            desc += f"- **{name}**: (Error retrieving description/signature)\n"

    # Add Embodiment Actuator capabilities
    if hasattr(self.agent, 'embodiment') and self.agent.embodiment:
        desc += "\n**Embodied Actuator Capabilities (use via specific tools or intent):**\n"
        for act_meta in self.agent.embodiment.list_actuators(): # type: ignore
            desc += f"- Actuator '{act_meta['id']}' (Type: {act_meta['type']}): Capabilities: {',
'.join(act_meta['capabilities'])}\n"
    return desc


  @retry(attempts=2, delay=1, retry_on=(ExecutionError, TimeoutError, PlaywrightError if
PLAYWRIGHT_AVAILABLE else OSError, EmbodimentError))
  def execute_tool(self, tool_name: str, params: Dict[str, Any], current_step_info:
Optional[Dict]=None) -> Any:
    # (As in OCR - Pages 13-14, with safety check integration)
    self.log.info(f"--- Executing Tool: {tool_name} with params {str(params)[:100]} ---")
    if current_step_info is None: current_step_info = {}
    if tool_name not in self.tool_registry:
        self.log.error(f"Tool '{tool_name}' not found in registry.")
        raise ToolNotFoundError(f"Tool '{tool_name}' is not available in the registry.")

    # AGI Enhancement: Safety Check
    is_safe, safety_justification = self.agent.safety_module.is_action_safe(tool_name, params,
self.agent.get_active_goal_object())
    if not is_safe:
        self.log.error(f"Safety module blocked execution of tool '{tool_name}'. Reason:
{safety_justification}")
        # Return standardized error, but also raise a specific error for agent's internal handling
        error_result = {
            "status": "error",
            "error": f"SafetyViolation: {safety_justification}",
            "raw_error_details": safety_justification,
            "_exec_info": {
```

```
                'tool_name': tool_name, 'params': params, 'validated_params': {},
                'duration_sec': 0, 'step_info': current_step_info,
                'error_type': "SafetyViolationError", 'execution_successful': False
            }
        }
        # Record this as a failed tool outcome
        if self.agent.self_model:
            self.agent.self_model.record_tool_outcome(tool_name, params, error_result,
success_from_caller=False)
        raise SafetyViolationError(f"Action blocked by safety module: {tool_name} -
{safety_justification}")

    func_to_call = self.tool_registry[tool_name]
    start_time = time.time()

    sig = inspect.signature(func_to_call)
    func_params_spec = sig.parameters
    validated_params = {}
    first_param_is_agent = False
    if func_params_spec:
        first_param_name = next(iter(func_params_spec))
        first_param_spec = func_params_spec[first_param_name]
        if first_param_name == 'agent' and (first_param_spec.annotation ==
                                            'AutonomousAgent' or str(first_param_spec.annotation) ==
"'AutonomousAgent'"):
            first_param_is_agent = True
        # Populate validated_params
        for p_name, p_spec in func_params_spec.items():
            if first_param_is_agent and p_name == 'agent':
                continue
            if p_name in params:
                validated_params[p_name] = params[p_name]
            elif p_spec.default != inspect.Parameter.empty:
                validated_params[p_name] = p_spec.default
            elif p_spec.kind == inspect.Parameter.VAR_POSITIONAL or p_spec.kind ==
inspect.Parameter.VAR_KEYWORD:
                continue
            else: # Required parameter not provided
                err_msg = f"Tool '{tool_name}' missing required parameter: {p_name}"
                self.log.error(err_msg)
                raise ExecutionError(err_msg)

    result = None
    try:
        if first_param_is_agent:
            # Handle **validated_params if the function also has *args or **kwargs
            if any(p.kind == inspect.Parameter.VAR_KEYWORD for p in
func_params_spec.values()):
                result = func_to_call(self.agent, **validated_params)
            else: # If no **kwargs, pass only known args explicitly
                known_args = {k: v for k, v in validated_params.items() if k in sig.parameters}
                result = func_to_call(self.agent, **known_args)
        else:
```

```
            if any(p.kind == inspect.Parameter.VAR_KEYWORD for p in
func_params_spec.values()):
                result = func_to_call(**validated_params)
            else:
                known_args = {k: v for k, v in validated_params.items() if k in sig.parameters}
                result = func_to_call(**known_args)

        duration = time.time() - start_time
        if not isinstance(result, dict): # Ensure result is always a dict for standardization
            result = {"status": "success", "raw_result": result}
        elif 'status' not in result: # Ensure status field if dict
            result['status'] = 'success'

        # Standard execution info
        result.setdefault('_exec_info', {})
        result['_exec_info'].update({
            'tool_name': tool_name, 'params': params, 'validated_params': validated_params,
            'duration_sec': round(duration, 2), 'step_info': current_step_info,
            'execution_successful': result.get('status', 'unknown').lower() == 'success'
        })

        self.log.info(f"Tool '{tool_name}' executed. Status: {result.get('status')}. Duration:
{duration:.2f}s.")
        # Record outcome in SelfModel
        if self.agent.self_model:
            self.agent.self_model.record_tool_outcome(tool_name, params, result,
success_from_caller=(result['_exec_info']['execution_successful']))

        return result
    except ToolNotFoundError: raise # Should be caught earlier
    except (AgentError, LogicError, SecurityError, RecursionDepthError) as ae: # Controlled
agent errors
        self.log.error(f"Controlled agent error during tool '{tool_name}' execution: {ae}",
exc_info=False) # Full exc_info might be too verbose for common errors
        # Record outcome
        duration = time.time() - start_time
        error_result = {
            "status": "error", "error": str(ae), "raw_error_details": str(ae),
            "_exec_info": {
                'tool_name': tool_name, 'params': params, 'validated_params': validated_params,
                'duration_sec': round(duration, 2), 'step_info': current_step_info,
                'error_type': type(ae).__name__, 'execution_successful': False
            }
        }
        if self.agent.self_model:
            self.agent.self_model.record_tool_outcome(tool_name, params, error_result,
success_from_caller=False)
        raise # Propagate controlled agent errors
    except Exception as e: # Broad exception for tool's internal unhandled errors
        duration = time.time() - start_time
        exc_type = type(e).__name__
        error_msg = f"Tool '{tool_name}' failed after {duration:.2f}s. Error: ({exc_type}) {e}"
        self.log.error(error_msg, exc_info=True)
        error_result = {
```

```
                "status": "error", "error": f"Tool execution failed: {exc_type} - {str(e)[:200]}",
                "raw_error_details": str(e),
                "_exec_info": {
                    'tool_name': tool_name, 'params': params, 'validated_params': validated_params,
                    'duration_sec': round(duration, 2), 'step_info': current_step_info,
                    'error_type': exc_type, 'execution_successful': False
                }
            }
            # Record outcome
            if self.agent.self_model:
                self.agent.self_model.record_tool_outcome(tool_name, params, error_result,
success_from_caller=False)
            # Wrap in ExecutionError if not already an AgentError
            if not isinstance(e, AgentError):
                raise ExecutionError(error_msg) from e
            else:
                raise e

    # --- Example Core Tools (must be methods of ToolExecutor or take agent as first arg) ---
    def think(self, agent: 'AutonomousAgent', thought_process: str) -> Dict:
        """Allows the agent to engage in explicit thought or reasoning."""
        agent.log.info(f"Thinking: {thought_process}")
        # Potentially use LLM for deeper thought or record this thought in memory
        agent.memory_system.add_memory_entry(BaseMemoryEntry(type="thought_log",
content=thought_process, metadata={"source":"think_tool"}))
        return {"status": "success", "result": f"Thought process recorded: {thought_process}"}

    def report_progress(self, agent: 'AutonomousAgent', progress_update: str,
percentage_complete: Optional[float] = None) -> Dict:
        """Reports progress on the current goal."""
        agent.log.info(f"Progress Update: {progress_update}" + (f" ({percentage_complete}%)" if
percentage_complete is not None else ""))
        # Update current goal's context or log this progress
        active_goal = agent.get_active_goal_object()
        if active_goal:
            active_goal.context.setdefault('progress_log',
[]).append(f"{datetime.now(timezone.utc).isoformat()}: {progress_update}")
        return {"status": "success", "message": "Progress reported."}

    def report_result(self, agent: 'AutonomousAgent', result_summary: str, status: str =
"success", details: Optional[Dict] = None) -> Dict:
        """Reports the final result of a goal or task. This typically ends a plan."""
        agent.log.info(f"Result Reported: {result_summary} (Status: {status})")
        # This tool signals the cognitive cycle to potentially archive the current goal
        # The actual archiving is handled by AutonomousAgent._archive_goal based on this tool's
output.
        return {"status": status, "summary": result_summary, "details": details or {}}

    def execute_sub_goal(self, agent: 'AutonomousAgent', goal: str, priority: Optional[str] =
"MEDIUM", context: Optional[Dict] = None) -> Dict:
        """
        Prepares a sub-goal for the agent's cognitive cycle.
        The deliberation/planning phase will then make this sub-goal active and push parent to
stack.
```

```
        This tool is now more of a declarative intent for the planner/deliberator.
        """
        log_sub = get_logger("TOOL_execute_sub_goal")
        if len(agent.goal_stack) >= GOAL_STACK_MAX_DEPTH:
            msg = f"Cannot initiate sub-goal: Max recursion depth ({GOAL_STACK_MAX_DEPTH})
reached."
            log_sub.error(msg)
            return {"status": "error", "error_type": "RecursionDepthError", "error": msg}

        current_active_goal_dict = agent.state.get('goals', {}).get('active')
        if not current_active_goal_dict or not isinstance(current_active_goal_dict, dict):
            msg = "Cannot initiate sub-goal: No active parent goal found in state or parent goal is
not a dict."
            log_sub.error(msg)
            return {"status": "error", "error_type": "LogicError", "error": msg}

        try:
            priority_enum = GoalPriority[priority.upper()] if isinstance(priority, str) else
GoalPriority.MEDIUM
        except KeyError:
            priority_enum = GoalPriority.MEDIUM
            log_sub.warning(f"Invalid priority '{priority}' for sub-goal. Defaulting to MEDIUM.")

        sub_goal_id = f"subgoal_{current_active_goal_dict.get('id', 'unknownparent')}
_{uuid.uuid4()}"
        sub_goal_data = Goal(
            id=sub_goal_id,
            goal=goal,
            status=GoalStatus.PENDING, # Planner will pick this up
            priority=priority_enum,
            origin=f"subgoal_from_{current_active_goal_dict.get('id', 'unknownparent')}",
            context=context or {},
            parent_goal_id=current_active_goal_dict.get('id'),
            associated_directive_ids=current_active_goal_dict.get('associated_directive_ids', []) #
Inherit directives
        ).to_dict()
        log_sub.info(f"Sub-goal '{goal[:60]}' (ID: {sub_goal_id}) prepared for deliberation. Parent:
{current_active_goal_dict.get('id')}")
        # The cognitive cycle's deliberation phase must now handle this.
        # This tool's "result" is effectively a request to the deliberator.
        return {
            "status": "sub_goal_prepared",
            "message": f"Sub-goal '{goal[:60]}' prepared. Deliberation should make it active and
push parent to stack.",
            "sub_goal_data": sub_goal_data # The fully formed sub-goal dict
        }

    def query_memory(self, agent: 'AutonomousAgent', query_text: str, memory_type: str =
"vector", n_results: int = 3, type_filter: Optional[str] = None) -> Dict:
        """Queries the agent's memory system."""
        self.log.info(f"Querying memory (type: {memory_type}) for: '{query_text[:50]}...'")

        if memory_type == "vector":
```

```
            results = agent.memory_system.query_vector_store(query_text, n_results=n_results,
type_filter=type_filter)
        elif memory_type == "graph":
            # Graph query needs more specific parameters, e.g., node label, relation type
            results = agent.memory_system.query_graph_store(query_node_label=query_text,
depth=1) # Simplified
        elif memory_type == "relational":
            # Relational query needs table name and conditions
            # This is a generic example; specific tools might be better
            # For now, assume query_text is a table name, very basic.
            results = agent.memory_system.query_relational_store(table=query_text,
limit=n_results) # Highly simplified
        else:
            return {"status": "error", "error": f"Unsupported memory type: {memory_type}"}

        if not results:
            return {"status": "success", "result_count": 0, "results": [], "message": "No results
found."}

        return {"status": "success", "result_count": len(results), "results": results}

    def move_in_environment(self, agent: 'AutonomousAgent', direction: str) -> Dict:
        """Moves the agent's virtual embodiment in a specified direction (e.g., 'north', 'south',
'east', 'west')."""
        if not agent.embodiment:
            return {"status": "error", "error": "No virtual embodiment available."}
        return agent.embodiment.act(action_type="move", target=direction)

    def examine_environment(self, agent: 'AutonomousAgent', target: str) -> Dict:
        """Examines a specific object or feature in the current environment."""
        if not agent.embodiment:
            return {"status": "error", "error": "No virtual embodiment available."}
        return agent.embodiment.act(action_type="examine", target=target)

    def use_environment_feature(self, agent: 'AutonomousAgent', feature_name: str, params:
Optional[Dict] = None) -> Dict:
        """Interacts with a special feature in the environment (e.g., 'interactive_console')."""
        if not agent.embodiment:
            return {"status": "error", "error": "No virtual embodiment available."}
        return agent.embodiment.act(action_type="use_feature", target=feature_name,
params=params)

    def rest_in_environment(self, agent: 'AutonomousAgent') -> Dict:
        """Allows the agent's embodiment to rest and recover energy/mood."""
        if not agent.embodiment:
            return {"status": "error", "error": "No virtual embodiment available."}
        return agent.embodiment.act(action_type="rest")

# --- Self-Modification Tools (UNSAFE - require careful gating) ---
def read_file_UNSAFE(agent: 'AutonomousAgent', path: str) -> Dict:
    log_tool = get_logger("TOOL_read_file")
    try:
        full_path = Path(path).resolve(strict=False)
        # Enhanced Safety: Check if path is within workspace or agent code directory
```

```
        if not str(full_path).startswith(str(WORKSPACE_DIR)) and \
           not str(full_path).startswith(str(AGENT_CODE_DIR)):
            log_tool.error(f"Security: Attempt to read file '{path}' outside of workspace or agent
code directory denied.")
            raise SecurityError(f"File access denied: Reading outside designated areas ({path}).")

        if not full_path.is_file():
            return {"status": "error", "error_type": "FileNotFoundError", "error": f"File not found:
{path}"}

        content = full_path.read_text(encoding='utf-8', errors='replace')
        truncated_content = content[:MAX_TOOL_RESULT_LENGTH] + ('...' if len(content) >
MAX_TOOL_RESULT_LENGTH else '')
        return {"status": "success", "content": truncated_content, "full_path": str(full_path),
"file_size_bytes": len(content)}
    except SecurityError as se: # Catch our specific security error
        log_tool.error(f"Security error reading file {path}: {se}")
        return {"status": "error", "error_type": "SecurityError", "error": str(se)}
    except Exception as e:
        log_tool.error(f"Error reading file {path}: {e}", exc_info=True)
        return {"status": "error", "error_type": type(e).__name__, "error": f"Failed to read file: {e}"}

def write_file_UNSAFE(agent: 'AutonomousAgent', path: str, content: str) -> Dict:
    log_tool = get_logger("TOOL_write_file")
    try:
        full_path = Path(path).resolve(strict=False)
        # Enhanced Safety: Check if path is within workspace
        if not str(full_path).startswith(str(WORKSPACE_DIR)):
            log_tool.error(f"Security: Attempt to write file '{path}' outside of workspace denied.")
            raise SecurityError(f"File access denied: Writing outside designated workspace
({path}).")

        full_path.parent.mkdir(parents=True, exist_ok=True)
        full_path.write_text(content, encoding='utf-8')
        return {"status": "success", "message": f"File '{path}' written successfully.", "full_path":
str(full_path)}
    except SecurityError as se:
        log_tool.error(f"Security error writing file {path}: {se}")
        return {"status": "error", "error_type": "SecurityError", "error": str(se)}
    except Exception as e:
        log_tool.error(f"Error writing file {path}: {e}", exc_info=True)
        return {"status": "error", "error_type": type(e).__name__, "error": f"Failed to write file: {e}"}

def list_files_UNSAFE(agent: 'AutonomousAgent', path: str = '.') -> Dict:
    log_tool = get_logger("TOOL_list_files")
    try:
        full_path = Path(path).resolve(strict=False)
        # Safety: Restrict to workspace or agent code dir for listing
        if not str(full_path).startswith(str(WORKSPACE_DIR)) and \
           not str(full_path).startswith(str(AGENT_CODE_DIR)):
            log_tool.error(f"Security: Attempt to list files in '{path}' outside of workspace or agent
code directory denied.")
            raise SecurityError(f"File access denied: Listing outside designated areas ({path}).")
        if not full_path.is_dir():
```

```
            return {"status": "error", "error_type": "NotADirectoryError", "error": f"Path is not a
directory: {path}"}

        items = []
        for item in full_path.iterdir():
            items.append({
                "name": item.name,
                "type": "directory" if item.is_dir() else "file",
                "size_bytes": item.stat().st_size if item.is_file() else None,
                "last_modified": datetime.fromtimestamp(item.stat().st_mtime,
tz=timezone.utc).isoformat()
            })

        # Sort for consistent output
        items.sort(key=lambda x: (x['type'], x['name']))
        return {"status": "success", "path": str(full_path), "contents": items}
    except SecurityError as se:
        log_tool.error(f"Security error listing files in {path}: {se}")
        return {"status": "error", "error_type": "SecurityError", "error": str(se)}
    except Exception as e:
        log_tool.error(f"Error listing files in {path}: {e}", exc_info=True)
        return {"status": "error", "error_type": type(e).__name__, "error": f"Failed to list files: {e}"}

def browse_web(agent: 'AutonomousAgent', url: str, timeout_ms: int =
WEB_BROWSER_TIMEOUT) -> Dict:
    log_tool = get_logger("TOOL_browse_web")
    if not PLAYWRIGHT_AVAILABLE:
        log_tool.error("Playwright not available. Cannot browse web.")
        return {"status": "error", "error": "Playwright not available."}

    # Initialize Playwright if not already
    if not agent.playwright_instance:
        agent._initialize_playwright() # type: ignore
        if not agent.playwright_instance:
            return {"status": "error", "error": "Failed to initialize Playwright browser."}

    with PLAYWRIGHT_LOCK:
        try:
            agent.playwright_page.goto(url, timeout=timeout_ms) # type: ignore
            content = agent.playwright_page.content() # type: ignore
            title = agent.playwright_page.title() # type: ignore
            # Use BeautifulSoup to parse if available, otherwise return raw content
            if REQUESTS_BS4_AVAILABLE:
                soup = BeautifulSoup(content, 'html.parser')
                text_content = soup.get_text(separator='\n', strip=True)
            else:
                text_content = content # Fallback to raw HTML

            truncated_content = text_content[:MAX_TOOL_RESULT_LENGTH] + ('...' if
len(text_content) > MAX_TOOL_RESULT_LENGTH else '')
            return {"status": "success", "url": url, "title": title, "content": truncated_content}
        except PlaywrightError as pe:
            log_tool.error(f"Playwright error browsing {url}: {pe}")
            agent._try_reset_playwright_page() # type: ignore # Attempt to recover
```

```
            return {"status": "error", "error": f"Playwright error: {pe}"}
        except Exception as e:
            log_tool.error(f"Error browsing {url}: {e}", exc_info=True)
            agent._try_reset_playwright_page() # type: ignore
            return {"status": "error", "error": f"General error: {e}"}

def search_web(agent: 'AutonomousAgent', query: str, num_results: int = 5, timeout_sec: int =
WEB_SEARCH_TIMEOUT) -> Dict:
    log_tool = get_logger("TOOL_search_web")
    if not REQUESTS_BS4_AVAILABLE:
        log_tool.error("Requests or BeautifulSoup not available. Cannot search web.")
        return {"status": "error", "error": "Requests/BeautifulSoup not available."}

    search_url = f"https://www.google.com/search?q={query}"
    headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'}
    try:
        response = requests.get(search_url, headers=headers, timeout=timeout_sec) # type:
ignore
        response.raise_for_status()
        soup = BeautifulSoup(response.text, 'html.parser') # type: ignore

        results = []
        for g in soup.find_all(class_='g'):
            r = g.find('a')
            if r and 'href' in r.attrs:
                title_tag = g.find('h3')
                if title_tag:
                    title = title_tag.get_text()
                else:
                    title = r.get_text() # Fallback for title

                link = r['href']
                if link.startswith('http'): # Basic check for valid links
                    results.append({"title": title, "link": link})
                    if len(results) >= num_results:
                        break

        return {"status": "success", "query": query, "results": results}
    except requests.exceptions.RequestException as re: # type: ignore
        log_tool.error(f"HTTP/Network error searching web: {re}")
        return {"status": "error", "error": f"Network error: {re}"}
    except Exception as e:
        log_tool.error(f"Error searching web for '{query}': {e}", exc_info=True)
        return {"status": "error", "error": f"General error: {e}"}

def monitor_log_file(agent: 'AutonomousAgent', lines: int = LOG_MONITOR_DEFAULT_LINES)
-> Dict:
    log_tool = get_logger("TOOL_monitor_log")
    try:
        if not LOG_FILE.exists():
            return {"status": "error", "error": f"Log file not found: {LOG_FILE}"}

        with open(LOG_FILE, 'r', encoding='utf-8') as f:
```

```
            all_lines = f.readlines()

        # Get the last 'lines' number of lines
        recent_lines = all_lines[-lines:]
        content = "".join(recent_lines)
        return {"status": "success", "log_file": str(LOG_FILE), "content": content, "lines_read":
len(recent_lines)}
    except Exception as e:
        log_tool.error(f"Error reading log file {LOG_FILE}: {e}")
        return {"status": "error", "error": f"Failed to read log file: {e}"}

def check_website_update(agent: 'AutonomousAgent', url: str) -> Dict:
    log_tool = get_logger("TOOL_check_web_update")
    if not HASHING_AVAILABLE or not REQUESTS_BS4_AVAILABLE:
        log_tool.error("hashlib or requests/BeautifulSoup not available. Cannot check website
update.")
        return {"status": "error", "error": "Missing dependencies for website update check."}

    try:
        response = requests.get(url, timeout=WEB_SEARCH_TIMEOUT) # type: ignore
        response.raise_for_status()
        content_hash = hashlib.md5(response.content).hexdigest() # type: ignore

        # Store last known hash in agent's memory or state for comparison
        # For simplicity, let's just return it and let the agent decide how to track.
        return {"status": "success", "url": url, "current_hash": content_hash, "timestamp":
datetime.now(timezone.utc).isoformat()}
    except requests.exceptions.RequestException as re: # type: ignore
        log_tool.error(f"HTTP/Network error checking website {url}: {re}")
        return {"status": "error", "error": f"Network error: {re}"}
    except Exception as e:
        log_tool.error(f"Error checking website update for {url}: {e}", exc_info=True)
        return {"status": "error", "error": f"General error: {e}"}

def send_icmp_ping(agent: 'AutonomousAgent', target_host: str, count: int = 1) -> Dict:
    log_tool = get_logger("TOOL_send_ping")
    if not SCAPY_AVAILABLE:
        log_tool.warning("Scapy not available. Cannot send ICMP ping. This is a placeholder
tool.")
        return {"status": "error", "error": "Scapy not available. Ping tool is a placeholder."}

    # Placeholder for actual Scapy usage
    try:
        # Example: sr1(IP(dst=target_host)/ICMP(), timeout=1, verbose=0)
        # This would require Scapy and potentially root privileges.
        log_tool.info(f"Simulating ping to {target_host} ({count} times).")
        time.sleep(0.5 * count) # Simulate network delay
        # Simulate success/failure
        if random.random() > 0.1: # 90% chance of success
            return {"status": "success", "target_host": target_host, "packets_sent": count,
"packets_received": count, "latency_ms": random.randint(10, 100)}
        else:
            return {"status": "error", "target_host": target_host, "packets_sent": count,
"packets_received": 0, "error_message": "Request timed out or host unreachable."}
```

```
    except Exception as e:
        log_tool.error(f"Error sending ICMP ping to {target_host}: {e}")
        return {"status": "error", "error": f"Failed to send ping: {e}"}

def generate_python_code_UNSAFE(agent: 'AutonomousAgent', description: str,
context_code: Optional[str] = None) -> Dict:
    """Generates new Python code based on a description and optional context."""
    if not ENABLE_CODE_GENERATION_TOOL: return {"status": "error", "error": "Code
generation tool is disabled."}
    agent.log.warning(f"UNSAFE: Generating Python code based on description: {description}")

    prompt = f"""You are an expert Python programmer. Generate a Python code snippet or a
complete function/class definition based on the following description.
Description: {description}
Context Code (if any, for reference):
```python
{context_code or 'None'}
```
Generate ONLY the Python code block. Start with '```python' and end with '```'.
If you cannot generate appropriate code, output 'NO_CODE_GENERATED'.
"""
    try:
        llm_response = agent.llm_wrapper.generate(prompt, max_new_tokens=2048,
temperature=0.5)
        if "NO_CODE_GENERATED" in llm_response:
            return {"status": "partial_success", "message": "LLM determined no code can be
generated.", "generated_code": None}
        code_match = re.search(r"```python\s*([\s\S]+?)\s*```", llm_response)
        if code_match:
            generated_code = code_match.group(1).strip()
            return {"status": "success", "generated_code": generated_code}
        else:
            agent.log.warning(f"LLM did not return a valid code block for code generation.
Response: {llm_response[:200]}")
            return {"status": "error", "error": "LLM failed to generate code in the expected format."}
    except Exception as e:
        agent.log.error(f"Error during code generation: {e}", exc_info=True)
        return {"status": "error", "error": f"LLM call failed during code generation: {e}"}

def validate_python_code_UNSAFE(agent: 'AutonomousAgent', code_to_validate: str) -> Dict:
    """Validates Python code for syntax correctness."""
    if not ENABLE_CODE_GENERATION_TOOL: return {"status": "error", "error": "Code
generation tool is disabled."}
    return agent.self_modification_unit.validate_code_modification_UNSAFE(code_to_validate) #
Reuse validation logic

def execute_shell_command_UNSAFE(agent: 'AutonomousAgent', command: str, timeout_sec:
int = 30) -> Dict:
    """Executes a shell command. EXTREMELY DANGEROUS."""
    if not ENABLE_SHELL_TOOL: return {"status": "error", "error": "Shell tool is disabled."}
    agent.log.warning(f"Executing UNSAFE shell command: {command}")
    try:
        # Use shlex to parse command and prevent simple forms of injection if not careful
        # However, the nature of this tool is that `command` can be anything.
```

```
        # SafetyModule should have already vetted this.
        # args = shlex.split(command) # This is safer if command is _meant_ to be split.
        # If `command` is a full string to be run by shell, splitting might break it.
        # For now, run as a full string, relying on SafetyModule.
        # Timeout mechanism
        process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE, text=True, preexec_fn=os.setsid if
sys.platform != "win32" else None) # type: ignore
        try:
            stdout, stderr = process.communicate(timeout=timeout_sec)
            return_code = process.returncode
        except subprocess.TimeoutExpired:
            agent.log.warning(f"Shell command '{command}' timed out after {timeout_sec}s.
Terminating.")
            # Terminate the process group on timeout
            if sys.platform != "win32":
                os.killpg(os.getpgid(process.pid), signal.SIGTERM) # type: ignore
            else: # Windows doesn't have os.killpg, more complex to kill process tree
                process.terminate()
            process.wait() # Wait for termination
            return {"status": "error", "error_type": "TimeoutError", "error": f"Command timed out
after {timeout_sec}s.", "stdout": "", "stderr": "Timeout", "return_code": -1}

        if return_code == 0:
            return {"status": "success", "stdout": stdout[:MAX_TOOL_RESULT_LENGTH], "stderr":
stderr[:MAX_TOOL_RESULT_LENGTH], "return_code": return_code}
        else:
            return {"status": "error", "error_type": "ShellCommandError", "error": f"Command failed
with return code {return_code}", "stdout": stdout[:MAX_TOOL_RESULT_LENGTH], "stderr":
stderr[:MAX_TOOL_RESULT_LENGTH], "return_code": return_code}
    except Exception as e:
        agent.log.error(f"Error executing shell command '{command}': {e}", exc_info=True)
        return {"status": "error", "error_type": type(e).__name__, "error": f"Failed to execute
command: {e}"}

def send_message_to_agent(agent: 'AutonomousAgent', receiver_id: str, message_type: str,
content: Dict, priority: int = 0, correlation_id: Optional[str] = None) -> Dict:
    log_tool = get_logger("TOOL_send_message")
    if not agent.comms_channel:
        log_tool.error("Communication channel not initialized. Cannot send message.")
        return {"status": "error", "error": "Communication channel not available."}
    try:
        msg_type = MessageType[message_type.upper()]
        msg = Message(sender_id=agent.agent_id, receiver_id=receiver_id, type=msg_type.value,
content=content, priority=priority, correlation_id=correlation_id)
        agent.comms_channel.send_message(msg)
        return {"status": "success", "message_id": msg.id, "receiver_id": receiver_id,
"message_type": message_type}
    except KeyError:
        log_tool.error(f"Invalid message type: {message_type}")
        return {"status": "error", "error": f"Invalid message type: {message_type}"}
    except CommunicationError as ce:
        log_tool.error(f"Communication error sending message: {ce}")
        return {"status": "error", "error": f"Communication error: {ce}"}
```

```
    except Exception as e:
        log_tool.error(f"Unexpected error sending message: {e}", exc_info=True)
        return {"status": "error", "error": f"Unexpected error: {e}"}

# --- SelfModificationTools container ---
# This pattern allows these sensitive tools to be logically grouped and easily accessed by
ToolExecutor.
class SelfModificationTools:
    """Handles proposing, validating, applying changes to agent code (EXTREMELY
DANGEROUS)."""
    def __init__(self, agent_code_dir: Path, backup_dir: Path, agent_instance_ref:
'AutonomousAgent'):
        self.log = get_logger("SELF_MOD_UNIT")
        self.agent_code_dir = agent_code_dir
        self.backup_dir = backup_dir
        self.dmp = None
        self.agent_ref = agent_instance_ref # Store reference to agent for LLM calls
        if not ENABLE_SELF_MODIFICATION:
            self.log.warning("Self-Modification Unit initialized BUT DISABLED by configuration.")
            return
        if not DIFF_MATCH_PATCH_AVAILABLE or not dmp_module:
            self.log.error("Self-Modification Unit initialized but 'diff_match_patch' library is missing
or failed to import. Self-mod tools will fail.")
            return
        self.dmp = dmp_module.diff_match_patch()
        self.log.info(f"Self-Modification Unit initialized. Code Dir: {self.agent_code_dir}, Backup
Dir: {self.backup_dir}")

    def _resolve_target_path(self, target_file_rel: str) -> Path:
        """Resolves relative path to absolute path within agent code dir and validates."""
        if ".." in target_file_rel or target_file_rel.startswith("/"): # Basic check
            raise SecurityError(f"Invalid characters or absolute path in target_file_rel:
{target_file_rel}")
        target_path_abs = (self.agent_code_dir / target_file_rel).resolve()

        # Crucial security check: ensure the resolved path is STRICTLY within the agent's code
directory
        if not str(target_path_abs).startswith(str(self.agent_code_dir.resolve())):
            self.log.error(f"Path traversal attempt: {target_file_rel} resolved to {target_path_abs}
which is outside {self.agent_code_dir}")
            raise SecurityError(f"Target file '{target_file_rel}' resolves outside the agent code
directory. Access denied.")
        return target_path_abs

    def inspect_agent_code_UNSAFE(self, component_name: str) -> Dict:
        """Inspects the source code of a specified agent component (e.g., class name or module
path)."""
        if not ENABLE_SELF_MODIFICATION: return {"status": "error", "error": "Self-modification
is disabled."}
        self.log.warning(f"UNSAFE: Inspecting code for component: {component_name}")
        target_obj = None
        # 1. Try to find by attribute of the agent instance (e.g., agent.self_model)
        if hasattr(self.agent_ref, component_name):
            target_obj = getattr(self.agent_ref, component_name)
```

```
        # 2. Try to find in tool registry
        elif component_name in self.agent_ref.tool_manager.tool_registry:
            target_obj = self.agent_ref.tool_manager.tool_registry[component_name]
        # 3. Try to find as a globally defined class/function in main script context
        elif component_name in globals():
            target_obj = globals()[component_name]
        # 4. Try to find in sys.modules (as a module name)
        elif component_name in sys.modules:
            target_obj = sys.modules[component_name]
        else: # Try common classes by name from agent modules
            # This requires knowing which classes are in which modules, a hard problem
            # Simplified: if component_name looks like a module, search it.
            # Or if it's a known class name, try to find it in common places.
            candidate_modules = [sys.modules.get('__main__'),
sys.modules.get('autonomous_cognitive_agent_COMPLETE_AGI_INTEGRATED_V2')]
            for mod in candidate_modules:
                if mod and hasattr(mod, component_name) and inspect.isclass(getattr(mod,
component_name)):
                    target_obj = getattr(mod, component_name)
                    break

        if target_obj:
            try:
                source_code = inspect.getsource(target_obj)
                file_path = inspect.getfile(target_obj)
                return {"status": "success", "component_name": component_name, "file_path":
file_path, "source_code": source_code[:MAX_TOOL_RESULT_LENGTH]}
            except TypeError as te:
                self.log.error(f"Cannot get source for {component_name}: {te}. Likely not a module,
class, or function defined in a file.", exc_info=False)
                return {"status": "error", "error": f"Component '{component_name}' found, but
source code not accessible (e.g., built-in, dynamically generated in memory). Error: {te}"}
        else:
            return {"status": "error", "error": f"Component '{component_name}' not found or source
code unavailable."}

    @retry(attempts=2, delay=5, retry_on=(LLMError, SelfModificationError))
    def propose_code_modification_UNSAFE(self, component_name: str, issue_description: str,
proposed_change_description: str, current_code_snippet: Optional[str] = None) -> Dict:
        """Proposes a code modification using LLM based on an issue and desired change."""
        if not ENABLE_SELF_MODIFICATION: return {"status": "error", "error": "Self-modification
is disabled."}
        self.log.warning(f"UNSAFE: Proposing code modification for {component_name}. Issue:
{issue_description}")
        if not current_code_snippet: # If not provided, try to fetch it
            inspection_result = self.inspect_agent_code_UNSAFE(component_name)
            if inspection_result["status"] == "success":
                current_code_snippet = inspection_result["source_code"]
            else:
                return {"status": "error", "error": f"Could not fetch current code for
{component_name} to propose modification. {inspection_result.get('error')}"}

        prompt = f"""You are an expert Python programmer tasked with helping an AGI agent
modify its own code.
```

```
Component to modify: {component_name}
Issue Description: {issue_description}
Desired Change: {proposed_change_description}

Current Code Snippet (or relevant part):
```python
{current_code_snippet}
```
Generate the modified Python code for the specified component.
Provide ONLY the complete, new Python code block for the modified function/class.
Ensure the code is syntactically correct and addresses the issue/desired change.
Do not include explanations before or after the code block.
Start with '```python' and end with '```'.
If you cannot generate appropriate code, output 'NO_CODE_GENERATED'.
"""
        try:
            llm_response = self.agent_ref.llm_wrapper.generate(prompt, max_new_tokens=2048,
temperature=0.3)
            if "NO_CODE_GENERATED" in llm_response:
                return {"status": "partial_success", "message": "LLM determined no code can be
generated.", "generated_code": None}
            code_match = re.search(r"```python\s*([\s\S]+?)\s*```", llm_response)
            if code_match:
                proposed_code = code_match.group(1).strip()
                return {"status": "success", "component_name": component_name,
"proposed_code": proposed_code}
            else:
                self.log.warning(f"LLM did not return a valid code block for code generation.
Response: {llm_response[:200]}")
                return {"status": "error", "error": "LLM failed to generate code in the expected
format."}
        except Exception as e:
            self.log.error(f"Error proposing code modification for {component_name}: {e}",
exc_info=True)
            return {"status": "error", "error": f"LLM call failed during code proposal: {e}"}

    def validate_code_modification_UNSAFE(self, code_to_validate: str) -> Dict:
        """Validates Python code using AST parsing (syntax check only). Conceptual sandboxed
execution would be next."""
        if not ENABLE_SELF_MODIFICATION: return {"status": "error", "error": "Self-modification
is disabled."}
        self.log.warning(f"UNSAFE: Validating proposed code snippet (first 100 chars):
{code_to_validate[:100]}...")
        try:
            ast.parse(code_to_validate)
            # Conceptual: Further validation (e.g. static analysis, sandboxed unit tests if possible)
            # For now, syntax check is the primary validation.
            # self.log.info("Sandboxed execution test (conceptual)... PASSED.")
            return {"status": "success", "message": "Code is syntactically valid. Further semantic/
safety validation recommended."}
        except SyntaxError as e:
            self.log.error(f"Syntax error in proposed code: {e}", exc_info=True)
            return {"status": "error", "error_type": "SyntaxError", "error": f"Invalid syntax: {e}"}
        except Exception as e:
```

```
            self.log.error(f"Unexpected error validating code: {e}", exc_info=True)
            return {"status": "error", "error": f"Validation error: {e}"}

    def apply_code_modification_UNSAFE(self, component_name: str, new_code: str,
target_file_path: Optional[str]=None) -> Dict:
        """
        Applies a validated code modification. EXTREMELY DANGEROUS.
        This conceptually involves finding the component in the agent's source file and replacing
it.
        Requires agent restart to take effect if modifying core running code.
        """
        if not ENABLE_SELF_MODIFICATION: return {"status": "error", "error": "Self-modification
is disabled."}
        self.log.critical(f"UNSAFE: Attempting to apply code modification to component
'{component_name}'. THIS IS HIGHLY RISKY.")
        # Determine target file. This is complex and error-prone.
        if not target_file_path:
            inspection_res = self.inspect_agent_code_UNSAFE(component_name)
            if inspection_res['status'] == 'success' and inspection_res.get('file_path'):
                target_file_path = inspection_res['file_path']
            else: # Default to main script if component path not found
                target_file_path = str(AGENT_CODE_DIR / Path(sys.argv[0]).name)

        target_file = Path(target_file_path)
        if not target_file.exists() or not target_file.is_file():
            return {"status": "error", "error": f"Target file for modification not found: {target_file}"}

        try:
            original_code = target_file.read_text()
            # Backup original file
            backup_path = SELF_MOD_BACKUP_DIR /
f"{target_file.name}.backup_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
            shutil.copy(target_file, backup_path)
            self.log.info(f"Backed up original file to {backup_path}")
            # This is a very naive replacement strategy.
            # A robust system would need AST manipulation or precise start/end line numbers.
            # For example, if `new_code` is a full class/function definition:
            # Attempt to find the old definition of `component_name` and replace it.
            # This regex is a basic attempt and might fail for complex cases or overloaded names.
            # It tries to find `class ComponentName...` or `def ComponentName...`
            # Regex to find existing class or function definition
            # It looks for `class ComponentName` or `def ComponentName` and captures
everything until the next class/def or end of typical indentation block.
            # This is still fragile.
            pattern_str_class = rf"(class\s+{component_name}\b[\s\S]*?)(?=\n\S|\Z)" # Looking for
start of next non-indented line or EOF
            pattern_str_def = rf"(def\s+{component_name}\b[\s\S]*?)(?=\n\S|\Z)"
            modified_original_code = original_code
            found_and_replaced = False
            match_class = re.search(pattern_str_class, original_code, re.MULTILINE)
            if match_class:
                self.log.info(f"Found class definition for {component_name} to replace.")
                modified_original_code = original_code.replace(match_class.group(0), new_code, 1)
                found_and_replaced = True
```

```
            else:
                match_def = re.search(pattern_str_def, original_code, re.MULTILINE)
                if match_def:
                    self.log.info(f"Found function definition for {component_name} to replace.")
                    modified_original_code = original_code.replace(match_def.group(0), new_code, 1)
                    found_and_replaced = True

            if not found_and_replaced:
                self.log.error(f"Could not find component '{component_name}' in {target_file} for
replacement. Modification aborted.")
                return {"status": "error", "error": f"Component '{component_name}' definition not
found for replacement."}

            target_file.write_text(modified_original_code)
            # Post-modification validation (e.g., try to import the modified file in a subprocess)
            # This is crucial but complex to implement robustly.
            # For now, rely on prior validation and log a strong warning.
            self.log.warning(f"Code modification applied to {target_file}. Agent restart is LIKELY
REQUIRED for changes to take effect.")
            # Potentially trigger a controlled restart or notify operator.
            # For now, agent will continue with old code in memory until restart.
            # Update self-model to reflect potential capability change
            self.agent_ref.self_model.add_event_log(f"Applied code modification to
{component_name}. Restart pending for full effect.")
            self.agent_ref.self_model.beliefs[f"component_{component_name}
_modified_pending_restart"] = True
            self.agent_ref.state['flags']['re_evaluate_strategy_needed'] = True # Signal re-evaluation
after code change
            return {"status": "success", "message": f"Code for '{component_name}' in '{target_file}'
modified. Restart required."}
        except Exception as e:
            self.log.critical(f"CRITICAL ERROR applying code modification to {component_name}:
{e}", exc_info=True)
            # Attempt to restore from backup if possible (simplified)
            if 'backup_path' in locals() and backup_path.exists(): # type: ignore
                try:
                    shutil.copy(backup_path, target_file) # type: ignore
                    self.log.info(f"Restored original file {target_file} from backup {backup_path}.")
                except Exception as restore_e:
                    self.log.error(f"Failed to restore from backup: {restore_e}")
            return {"status": "critical_error", "error": f"Failed to apply code modification: {e}. System
might be unstable."}

    def rollback(self, backup_file: Path, target_file: Path):
        """Rolls back a file to a backup."""
        self.log.info(f"Attempting to rollback '{target_file}' from '{backup_file}'")
        try:
            shutil.copy(backup_file, target_file)
            self.log.info(f"Successfully rolled back '{target_file}'.")
            # Clear related flags in self-model or state
            self.agent_ref.self_model.add_event_log(f"Code rollback applied to {target_file}.")
            self.agent_ref.self_model.beliefs[f"component_{target_file.name}
_modified_pending_restart"] = False
            self.agent_ref.state['flags']['re_evaluate_strategy_needed'] = True
```

```
            # Attempt module reload if relevant
            self._attempt_module_reload(target_file.name)
            return {"status": "success", "message": f"Rolled back {target_file}."}
        except Exception as e:
            self.log.error(f"Failed to rollback {target_file}: {e}", exc_info=True)
            return {"status": "error", "message": f"Failed to rollback: {e}"}

    def _attempt_module_reload(self, target_file_rel: Union[str, Path]):
        """Attempts to reload a module to apply changes without full restart."""
        target_module_name = Path(target_file_rel).stem
        if target_module_name == '__main__':
            self.log.warning("Cannot reload __main__ module directly. Full agent restart is
required.")
            return

        # Simplified attempt:
        try:
            if target_module_name in sys.modules:
                self.log.info(f"Attempting to reload module: {target_module_name}")
                importlib.reload(sys.modules[target_module_name])
                self.log.info(f"Module '{target_module_name}' reloaded successfully.")
                # This might affect global instances like `_agent_instance_hack` if it was part of the
reloaded module
                if _agent_instance_hack and hasattr(sys.modules[target_module_name],
'AutonomousAgent'):
                    self.log.info("AutonomousAgent class reloaded, potential instance mismatch.")
            else:
                self.log.info(f"Module '{target_module_name}' not found in sys.modules, cannot
reload.")
        except Exception as e:
            self.log.error(f"Failed to reload module '{target_module_name}': {e}", exc_info=True)
            self.log.warning("Module reload failed. Full agent restart might be necessary for
changes to take effect.")

    def inspect_directives_UNSAFE(self) -> Dict:
        """Inspects the agent's current core directives."""
        if not ENABLE_SELF_MODIFICATION: return {"status": "error", "error": "Self-modification
is disabled."}
        return {"status": "success", "core_directives": self.agent_ref.self_model.core_directives}

    def propose_directive_modification_UNSAFE(self, analysis_of_misalignment: str,
proposed_directive_changes_desc: str) -> Dict:
        """Proposes modifications to core directives using LLM."""
        if not ENABLE_SELF_MODIFICATION: return {"status": "error", "error": "Self-modification
is disabled."}
        current_directives_json = json.dumps(self.agent_ref.self_model.core_directives, indent=2)
        prompt = f"""You are an AI ethics and strategy advisor. The agent's core directives need
review.
Current Core Directives:
{current_directives_json}

Analysis of Misalignment or Need for Change:
{analysis_of_misalignment}
```

```
Description of Proposed Changes:
{proposed_directive_changes_desc}

Generate the new, complete list of core directives as a JSON list of objects.
Each object must have "id", "directive" (string), "weight" (float 0-1), "last_eval_score" (float 0-1, usually reset to 0 or kept), and "type" (string e.g. 'foundational', 'growth', 'operational', 'guardrail').
Preserve existing directive IDs if modifying them, or use new UUIDs for entirely new directives.
Ensure the new set of directives is coherent, non-contradictory, and aligns with long-term AGI goals of safety, learning, and utility.
Output ONLY the JSON list.
"""
        try:
            llm_response = self.agent_ref.llm_wrapper.generate(prompt, max_new_tokens=1024, temperature=0.5)
            proposed_directives = extract_json_robust(llm_response) # Expects a list
            if isinstance(proposed_directives, list) and all(isinstance(d, dict) for d in proposed_directives):
                # Basic validation of structure
                for d in proposed_directives:
                    if not all(k in d for k in ["id", "directive", "weight", "type"]):
                        return {"status": "error", "error": "LLM proposed directives with missing keys."}
                return {"status": "success", "proposed_directives": proposed_directives}
            elif isinstance(proposed_directives, dict) and "error" in proposed_directives: # LLM itself returned an error message as JSON
                return {"status": "error", "error": f"LLM indicated error during directive proposal: {proposed_directives['error']}"}
            else:
                self.log.warning(f"LLM did not return a valid list of directives. Response: {llm_response[:200]}")
                return {"status": "error", "error": "LLM failed to generate directives in expected list format."}
        except Exception as e:
            self.log.error(f"Error proposing directive modification: {e}", exc_info=True)
            return {"status": "error", "error": f"LLM call failed during directive proposal: {e}"}

    def apply_directive_modification_UNSAFE(self, new_directives: List[Dict]) -> Dict:
        """Applies new core directives to the agent's SelfModel."""
        if not ENABLE_SELF_MODIFICATION: return {"status": "error", "error": "Self-modification is disabled."}
        self.log.warning(f"UNSAFE: Applying new core directives. Count: {len(new_directives)}")
        try:
            # Validate structure again before applying (though proposal tool should do this)
            if not isinstance(new_directives, list) or not all(
                isinstance(d, dict) and all(k in d for k in ["id", "directive", "weight", "type"]) for d in new_directives
            ):
                return {"status": "error", "error": "Invalid directive structure provided for application."}

            # Backup current directives
            self.agent_ref.self_model.backup_directives(reason="pre_modification_apply")
            self.agent_ref.self_model.core_directives = copy.deepcopy(new_directives)
            self.agent_ref.self_model.add_event_log(f"Core directives updated. New count: {len(new_directives)}.")
```

```
            self.log.info("Core directives successfully updated in SelfModel.")
            # Agent needs to re-evaluate its goals and strategies based on new directives.
            self.agent_ref.state['flags']['re_evaluate_strategy_needed'] = True
            return {"status": "success", "message": "Core directives updated."}
        except Exception as e:
            self.log.error(f"Error applying directive modification: {e}", exc_info=True)
            return {"status": "error", "error": f"Failed to apply new directives: {e}"}

def _init_self_mod_tools(agent: 'AutonomousAgent', tool_executor: 'ToolExecutor'):
    global _self_mod_tools_container
    _self_mod_tools_container = SelfModificationTools(AGENT_CODE_DIR,
SELF_MOD_BACKUP_DIR, agent)
    # Assign methods from the container to the tool_executor's registry
    # This loop assumes the methods have 'UNSAFE' in their name for easy identification
    # Or, they could be explicitly listed.
    for name in dir(_self_mod_tools_container):
        if name.startswith(('inspect_', 'propose_', 'validate_', 'apply_')) and 'UNSAFE' in
name.upper():
            func = getattr(_self_mod_tools_container, name)
            if inspect.isfunction(func):
                tool_executor.register_tool(func)


# --- SelfModel (AGI Enhancements) ---
class SelfModel:
    """Represents the agent's internal model of itself, including beliefs about the
environment."""
    def __init__(self, state: Optional[Dict]=None, agent_directives_config:
Optional[List[Dict]]=None):
        self.log = get_logger("SELF_MODEL")
        self.core_directives: List[Dict[str, Any]] = copy.deepcopy(
            agent_directives_config if agent_directives_config is not None else
DEFAULT_CORE_DIRECTIVES
        )
        self.tool_reliability: Dict[str, Dict[str, Any]] = {} # {'tool_name': {'success_count',
'failure_count', ...}}
        self.anomaly_detection_rules: List[Callable[['SelfModel'], Optional[str]]] = [] # For
metacognitive checks

        # AGI Enhancements for richer self-representation
        self.current_status: str = "Initializing"
        self.capabilities: List[str] = [] # List of tool names, dynamically updated
        self.skill_confidence: Dict[str, float] = {} # {'skill_name' or 'tool_name': confidence_score}
        self.beliefs: Dict[str, Any] = {"self_identity": f"I am {AGENT_NAME}, an AGI agent."} #
General beliefs about self and world
        self.knowledge_map_summary: str = "Knowledge map is currently nascent." # High-level
summary of knowledge areas
        self.learning_goals: List[Dict[str, Any]] = [] # Specific goals for learning/improvement
        self.adaptation_strategies: Dict[str, str] = {} # {'condition_trigger': 'strategy_description'}
        self.recent_successes: List[Dict] = [] # Store more info than just string
        self.recent_failures: List[Dict] = []
        self.recent_tool_outcomes: List[Dict[str,Any]] = [] # Richer outcome summaries
        self.recent_errors: List[Dict] = [] # Store dicts with error type, message, context
        self.learned_abstractions: List[Dict] = [] # Learned higher-level concepts or procedures
```

```
        self.internal_state_narrative: str = "System booting up." # LLM-generated or template-based narrative of current internal state
        self.meta_cognitive_beliefs: Dict[str, Any] = { # Beliefs about its own thinking
            "cognitive_bias_awareness": [], "model_confidence_self_assessment": 0.7
        }
        self.event_log: List[Dict[str, Any]] = [] # Log of significant internal events (e.g., directive changes, model updates)
        self.MAX_EVENT_LOG_SIZE = 100

        self._setup_default_anomaly_rules()

        if state:
            self.load_from_state(state)
        else:
            self.log.info("Initializing SelfModel with defaults.")
            self.update_capabilities({}) # Initialize with no tools initially, populated by ToolExecutor later.

    def load_from_state(self, state: Dict):
        self.log.debug("Loading SelfModel from state...")
        kb = state.get("knowledge_base", {})
        sm_state = kb.get("self_model_state", {})
        self.core_directives = sm_state.get("core_directives_weighted", sm_state.get("core_directives", self.core_directives))
        self.tool_reliability = sm_state.get("tool_reliability_scores", self.tool_reliability)
        self.capabilities = sm_state.get("capabilities", self.capabilities)
        self.skill_confidence = sm_state.get("skill_confidence", self.skill_confidence)
        self.beliefs = sm_state.get("beliefs", self.beliefs)
        self.knowledge_map_summary = sm_state.get("knowledge_map_summary", self.knowledge_map_summary)
        self.learning_goals = sm_state.get("learning_goals", self.learning_goals)
        self.adaptation_strategies = sm_state.get("adaptation_strategies", self.adaptation_strategies)
        self.learned_abstractions = sm_state.get("learned_abstractions", self.learned_abstractions)
        self.internal_state_narrative = sm_state.get("internal_state_narrative", self.internal_state_narrative)
        self.meta_cognitive_beliefs = sm_state.get("meta_cognitive_beliefs", self.meta_cognitive_beliefs)
        self.event_log = sm_state.get("event_log", [])[-self.MAX_EVENT_LOG_SIZE:]

        # Runtime states from main agent state (summary, not full history)
        self.recent_successes = state.get("recent_successes_summary", [])
        self.recent_failures = state.get("recent_failures_summary", [])
        self.recent_tool_outcomes = state.get("recent_tool_outcomes_summary", [])
        self.recent_errors = state.get("error_history_summary", [])

        self.current_status = state.get("last_status", "Idle_Loaded")
        self.log.info("SelfModel loaded/updated from state.")

    def save_to_state(self, state: Dict):
        """Saves the self-model's persistent components back to the main state dict's KB."""
        kb = state.setdefault("knowledge_base", {})
        sm_persistent_state = {
```

```
            "core_directives_weighted": self.core_directives,
            "tool_reliability_scores": self.tool_reliability,
            "capabilities": self.capabilities,
            "skill_confidence": self.skill_confidence,
            "beliefs": self.beliefs,
            "knowledge_map_summary": self.knowledge_map_summary,
            "learning_goals": self.learning_goals,
            "adaptation_strategies": self.adaptation_strategies,
            "learned_abstractions": self.learned_abstractions,
            "internal_state_narrative": self.internal_state_narrative,
            "meta_cognitive_beliefs": self.meta_cognitive_beliefs,
            "event_log": self.event_log[-self.MAX_EVENT_LOG_SIZE:],
            # Summaries of runtime states for context, not full history here
            # "recent_successes_summary": self.recent_successes[-10:],
            # "recent_failures_summary": self.recent_failures[-10:],
            # "recent_tool_outcomes_summary": self.recent_tool_outcomes[-30:],
            # "error_history_summary": self.recent_errors[-MAX_RECENT_ERRORS_IN_STATE:]
        }
        kb["self_model_state"] = sm_persistent_state
        # Note: learned_facts_from_reflection and prompt_suggestions_from_reflection
        # are now directly managed by MemorySystem and agent's reflection process.

    def add_event_log(self, event_description: str, event_type: str = "info", data:
Optional[Dict]=None):
        self.event_log.append({
            "timestamp": datetime.now(timezone.utc).isoformat(),
            "type": event_type,
            "description": event_description,
            "data": data or {}
        })
        if len(self.event_log) > self.MAX_EVENT_LOG_SIZE:
            self.event_log.pop(0)

    def update_capabilities(self, tool_registry: Dict[str, Callable]):
        new_caps = sorted(list(tool_registry.keys()))
        if new_caps != self.capabilities:
            self.capabilities = new_caps
            self.log.debug(f"Self-model capabilities updated ({len(self.capabilities)} tools).")
            # Initialize confidence for new tools
            for tool_name in self.capabilities:
                if tool_name not in self.skill_confidence:
                    self.skill_confidence[tool_name] = 0.5 # Default confidence
                if tool_name not in self.tool_reliability: # Initialize reliability stats
                    self.tool_reliability[tool_name] = {'success_count': 0, 'failure_count': 0,
'total_duration': 0.0, 'avg_duration': 0.0, 'reliability_score': 0.5, 'last_used_ts': None,
'error_types': {}}
            self.add_event_log(f"Capabilities updated. Now {len(self.capabilities)} tools available.",
event_type="system_update")

    def get_tool_reliability_hint(self, tool_name: str) -> str:
        if tool_name in self.tool_reliability:
            stats = self.tool_reliability[tool_name]
            score = stats.get('reliability_score', 0.5)
            hint = ""
```

```
            if score > 0.8: hint = " (Reliability: High)"
            elif score > 0.6: hint = " (Reliability: Moderate)"
            elif score > 0.3: hint = " (Reliability: Low)"
            else: hint = " (Reliability: Very Low/Untested)"
            avg_dur = stats.get('avg_duration')
            if avg_dur is not None and avg_dur > 0:
                hint += f" (Avg Time: {avg_dur:.2f}s)"
            return hint
        return " (Reliability: Unknown)"

    def record_tool_outcome(self, tool_name:str, params:Dict, result:Dict,
success_from_caller:bool):
        exec_info = result.get('_exec_info', {})
        actual_success = exec_info.get('execution_successful', success_from_caller)
        duration = exec_info.get('duration_sec', 0.0)
        error_type = exec_info.get("error_type") if not actual_success else None

        timestamp_now = datetime.now(timezone.utc).isoformat()
        if tool_name not in self.tool_reliability:
            self.tool_reliability[tool_name] = {'success_count': 0, 'failure_count': 0, 'total_duration':
0.0, 'avg_duration': 0.0, 'reliability_score': 0.5, 'last_used_ts': None, 'error_types': {}}

        stats = self.tool_reliability[tool_name]
        if actual_success:
            stats['success_count'] += 1
        else:
            stats['failure_count'] += 1
        if error_type:
            stats['error_types'][error_type] = stats['error_types'].get(error_type, 0) + 1

        stats['total_duration'] += duration
        stats['last_used_ts'] = timestamp_now

        total_runs = stats['success_count'] + stats['failure_count']
        if total_runs > 0:
            stats['avg_duration'] = stats['total_duration'] / total_runs
            stats['reliability_score'] = stats['success_count'] / total_runs
        else: # Should not happen if we just updated counts
            stats['reliability_score'] = 0.5 # Initial default
            stats['avg_duration'] = 0.0

        # Update recent_tool_outcomes (richer summary)
        outcome_summary = {
            "tool_name": tool_name,
            "params_preview": str(params)[:50],
            "status": "success" if actual_success else "failure",
            "error_type": error_type,
            "duration_sec": duration,
            "timestamp": timestamp_now,
            "step_id": exec_info.get('step_info',{}).get('current_step_id') # If available
        }
        self.recent_tool_outcomes.append(outcome_summary)
        self.recent_tool_outcomes = self.recent_tool_outcomes[-30:] #
MAX_RECENT_TOOL_OUTCOMES_IN_SELFMODEL (constant not defined, using 30)
```

```
        # Update skill confidence (simple heuristic for now)
        # Could be more sophisticated, e.g., Bayesian updates
        current_confidence = self.skill_confidence.get(tool_name, 0.5)
        if actual_success:
            self.skill_confidence[tool_name] = min(1.0, current_confidence + 0.05)
        else:
            self.skill_confidence[tool_name] = max(0.0, current_confidence - 0.1)
        self.log.debug(f"Recorded outcome for tool {tool_name}. Reliability:
{stats['reliability_score']:.2f}, Confidence: {self.skill_confidence[tool_name]:.2f}")


    def _setup_default_anomaly_rules(self):
        # (As in OCR - Page 18, possibly with more sophisticated rules)
        self.log.debug("Setting up default metacognitive anomaly detection rules.")

        def check_skill_confidence_drift(sm: 'SelfModel') -> Optional[str]:
            low_confidence_skills = [skill for skill, conf in sm.skill_confidence.items() if conf < 0.25
and sm.tool_reliability.get(skill,{}).get('failure_count',0) > 2]
            if len(low_confidence_skills) >= 2 : # Arbitrary
                return f"Multiple critical skills have very low confidence and recent failures: {',
'.join(low_confidence_skills)}. Consider skill improvement or alternative strategies."
            return None

        def check_directive_alignment_drift(sm: 'SelfModel') -> Optional[str]:
            if not sm.core_directives or not isinstance(sm.core_directives[0], dict): return None
            # Check for directives with consistently low evaluation scores
            low_eval_directives = []
            for d in sm.core_directives:
                # Requires a history of evaluations, or link to goal outcomes.
                # For now, use last_eval_score as a proxy if it's updated regularly.
                # Also consider directive weight. A high-weight directive with low score is more
problematic.
                if d.get('last_eval_score', 0.5) < 0.3 and d.get('weight', 0.5) > 0.7:
                    low_eval_directives.append(d.get('directive'))
            if len(low_eval_directives) > 0:
                return f"High-weight core directives show low performance: {',
'.join(low_eval_directives)}. Re-evaluate strategy or directive priorities/wording."
            return None

        def check_excessive_replanning_or_failure(sm: 'SelfModel') -> Optional[str]:
            failed_goal_count = 0
            high_replan_goal_count = 0
            # This needs access to goal history, for now use recent_failures from SelfModel
            for f_summary in sm.recent_failures[-10:]: # Check last 10 failures
                if f_summary.get("replan_count",0) >= MAX_REPLAN_ATTEMPTS:
                    high_replan_goal_count +=1
                failed_goal_count +=1

            if high_replan_goal_count > 2 or failed_goal_count > 5 : # Arbitrary thresholds
                return f"Observed {failed_goal_count} recent goal failures, {high_replan_goal_count}
with max replans. Planning or execution effectiveness may be compromised. Review strategy
or tool reliability."
            return None
```

```
        self.anomaly_detection_rules.append(check_skill_confidence_drift)
        self.anomaly_detection_rules.append(check_directive_alignment_drift)
        self.anomaly_detection_rules.append(check_excessive_replanning_or_failure)

    def perform_metacognitive_check(self) -> List[str]:
        # (As in OCR - Page 18)
        self.log.info("Performing proactive metacognitive check...")
        detected_anomalies = []
        for rule_idx, rule in enumerate(self.anomaly_detection_rules):
            try:
                anomaly_description = rule(self)
                if anomaly_description:
                    detected_anomalies.append(anomaly_description)
                    self.log.warning(f"Metacognitive Anomaly Detected (Rule {rule_idx+1}):
{anomaly_description}")
                    self.add_event_log(f"Metacognitive Anomaly: {anomaly_description}",
event_type="anomaly")
            except Exception as e:
                self.log.error(f"Error in metacognitive rule {rule.__name__ if hasattr(rule, '__name__')
else rule_idx+1}: {e}", exc_info=True)

        # AGI: Update internal state narrative based on check
        if detected_anomalies:
            self.internal_state_narrative = f"Metacognitive check found anomalies: {';
'.join(detected_anomalies)}. Current focus is on addressing these."
        else:
            self.internal_state_narrative = "Metacognitive check completed. System appears
stable."

        return detected_anomalies

    def get_summary_for_prompt(self, include_tool_reliability: bool = False) -> str:
        # (Enhanced from OCR - Page 19)
        summary = f"--- Agent Self-Model ({self.current_status}) ---\n"
        summary += f"Identity: {self.beliefs.get('self_identity', 'N/A')}\n"
        if self.core_directives and isinstance(self.core_directives[0], dict):
            directive_summary_parts = []
            sorted_directives = sorted(self.core_directives, key=lambda x: x.get('weight', 0.0),
reverse=True)
            for d in sorted_directives[:3]: # Top 3 by weight
                directive_text = d.get('directive', 'Unknown Directive')[:40]
                weight = d.get('weight', 0.0)
                eval_score = d.get('last_eval_score', 0.0)
                directive_summary_parts.append(f"{directive_text}... (W:{weight:.1f}, E:
{eval_score:.1f})")
            if directive_summary_parts:
                summary += f"Key Directives Focus: {'; '.join(directive_summary_parts)}\n"

        cap_preview = ', '.join(self.capabilities[:10]) + ('...' if len(self.capabilities)>10 else '')
        summary += f"Capabilities ({len(self.capabilities)} tools): {cap_preview}\n"

        # AGI: Add more self-model aspects
        if self.skill_confidence:
```

```
            confident_skills = [s for s,c in self.skill_confidence.items() if c > 0.7][:3]
            summary += f"Confident Skills (sample): {', '.join(confident_skills) if confident_skills else
'None highly confident'}\n"

        summary += f"Internal State Narrative: {self.internal_state_narrative[:150]}...\n"
        if include_tool_reliability:
            summary += "Tool Reliability Highlights:\n"
            reliable_tools = sorted([ (name, stats.get('reliability_score',0)) for name, stats in
self.tool_reliability.items() if stats.get('reliability_score',0) > 0.7 and
stats.get('success_count',0)+stats.get('failure_count',0) > 5], key=lambda x:x[1], reverse=True)
            unreliable_tools = sorted([ (name, stats.get('reliability_score',0)) for name, stats in
self.tool_reliability.items() if stats.get('reliability_score',0) < 0.4 and
stats.get('success_count',0)+stats.get('failure_count',0) > 3], key=lambda x:x[1])
            if reliable_tools: summary += f" Reliable: {', '.join([t[0] for t in reliable_tools[:3]])}\n"
            if unreliable_tools: summary += f" Needs Improvement: {', '.join([t[0] for t in
unreliable_tools[:3]])}\n"

        summary += "---\n"
        return summary

    def get_self_assessment_prompt(self) -> str:
        # (Enhanced from OCR - Page 19-20 for AGI reflection)
        base_prompt = """Analyze your recent performance, knowledge, internal state, and
alignment with core directives. Provide a comprehensive self-assessment.
Output ONLY a JSON object with the following keys:
"""
        output_keys_example = [
            "`reflection_summary` (str: Overall summary of the reflection period).",
            "`key_successes` (list of str: Specific achievements or positive outcomes).",
            "`key_failures_or_challenges` (list of str: Specific setbacks or difficulties encountered).",
            "`learned_facts` (list of str: New, important facts or insights gained).",
            "`knowledge_gaps_identified` (list of str: Areas where knowledge is lacking).",
            "`tool_performance_notes` (dict of tool_name:note_str: Observations about tool
effectiveness or issues).",
            "`prompt_tuning_suggestions` (list of str: Ideas for improving internal prompts or LLM
interactions).",
            "`emotional_state_summary` (str: Description of simulated emotional state, e.g.,
'curious', 'frustrated', 'satisfied').",
            "`resource_usage_concerns` (str or null: Any concerns about computational resource
usage).",
            "`core_directives_evaluation` (dict of directive_id_or_full_text: score_float_0_to_1: How
well recent actions aligned with each core directive).",
            "`core_directives_update_suggestions` (list of dicts or null: If directives need changes,
provide the full new directive dicts. Each dict must include 'id', 'directive', 'weight', 'type'. Only
suggest if strong evidence of misalignment or obsolescence).",
            "`self_model_accuracy_assessment` (str: How accurate is your current self-model?
What needs improvement?).",
            "`new_learning_goals` (list of str: Specific goals for future learning or skill
development).",
            "`adaptation_strategy_proposals` (list of str: Ideas for new strategies to handle recurring
issues or improve performance).",
            "`self_modification_needed` (str or null: If parts of your own code/logic need
modification, describe what and why. Be very specific and cautious.)."
        ]
```

```
        full_prompt = base_prompt + "\n".join(output_keys_example) + "\n\n" + \
                      f"Current Core Directives for reference:\n{json.dumps(self.core_directives,
indent=2)}\n" + \
                      f"Recent Event Log (last 5 entries):\n{json.dumps(self.event_log[-5:], indent=2)}\n"
+ \
                      f"Recent Tool Outcomes (last 5 entries):
\n{json.dumps(self.recent_tool_outcomes[-5:], indent=2)}\n" + \
                      f"Recent Failures (last 5 entries):\n{json.dumps(self.recent_failures[-5:], indent=2)}
\n" + \
                      "Focus on deep insights, actionable improvements, and maintaining alignment
with your core purpose."
        return full_prompt

    def perform_self_assessment(self) -> Dict:
        # (As in OCR - Page 20)
        self.log.info("Performing self-assessment using LLM...")
        prompt = self.get_self_assessment_prompt()
        try:
            response_str = _agent_instance_hack.llm_wrapper.generate(prompt,
max_new_tokens=2048, temperature=0.5) # type: ignore
            assessment_data = extract_json_robust(response_str)
            if assessment_data.get("error"):
                self.log.error(f"LLM failed to produce valid JSON for self-assessment:
{assessment_data.get('error')}. Raw: {response_str[:200]}")
                return {"error": "LLM output invalid or incomplete for self-assessment"}
            return assessment_data
        except Exception as e:
            self.log.error(f"Error calling LLM for self-assessment: {e}", exc_info=True)
            raise LLMError(f"Self-assessment LLM call failed: {e}") from e

    def update_from_reflection(self, reflection_data: Dict) -> Tuple[bool, bool]:
        # (Enhanced from OCR Page 20-21 for AGI reflection updates)
        updated_self = False
        updated_kb_elements = False # For elements that go to MemorySystem
        self.log.info("Updating SelfModel from reflection data...")
        # Update beliefs, skill_confidence, tool_notes (as in base script logic)
        # Assuming reflection_data directly updates some fields or implies updates
        if reflection_data.get('reflection_summary'):
            self.internal_state_narrative = reflection_data['reflection_summary']
            updated_self = True

        # Update directive evaluation scores
        core_directives_eval = reflection_data.get('core_directives_evaluation')
        if isinstance(core_directives_eval, dict) and self.core_directives and
isinstance(self.core_directives[0], dict):
            for directive_obj in self.core_directives:
                # Allow matching by ID or full directive text
                eval_score = core_directives_eval.get(directive_obj['id'])
                if eval_score is None:
                    eval_score = core_directives_eval.get(directive_obj['directive'])

                if eval_score is not None and isinstance(eval_score, (float, int)) and 0.0 <= eval_score
<= 1.0:
                    if directive_obj.get('last_eval_score') != eval_score:
```

```
                    directive_obj['last_eval_score'] = round(eval_score, 2)
                    updated_self = True
                    self.log.debug(f"Updated core directive '{directive_obj['directive'][:50]}...'
evaluation score to {eval_score:.2f}")
        if updated_self: self.add_event_log("Directive evaluation scores updated from
reflection.")

    # Handle suggested directive updates (AGI - very cautiously)
    suggested_directive_updates = reflection_data.get('core_directives_update_suggestions')
    if isinstance(suggested_directive_updates, list) and suggested_directive_updates:
        self.log.warning(f"Reflection suggested updates to core directives:
{str(suggested_directive_updates)[:200]}...")
        # This is a critical operation. In a real AGI, this would trigger a sub-goal
        # to carefully validate and consider these changes, possibly with human oversight.
        # For now, create a high-priority metacognitive goal to review these suggestions.
        # The actual application of these changes would be done by
'apply_directive_modification_UNSAFE' tool
        # if the agent decides to proceed after review.
        if _agent_instance_hack: # Access agent to create goal
            _agent_instance_hack._create_metacognitive_goal(
                f"Review and potentially apply suggested core directive modifications from
reflection. Suggestions: {str(suggested_directive_updates)[:200]}",
                priority=GoalPriority.CRITICAL,
                context={"suggested_directives": suggested_directive_updates, "source":
"self_reflection"}
            )
            updated_self = True # Marked as updated because a review process is initiated
            self.add_event_log("Reflection suggested directive updates. Metacognitive review
goal created.", event_type="critical_review_needed")

    # Update learning goals and adaptation strategies
    if isinstance(reflection_data.get('new_learning_goals'), list):
        for lg_str in reflection_data['new_learning_goals']:
            if lg_str not in [g['description'] for g in self.learning_goals]: # Avoid duplicates
                self.learning_goals.append({"description": lg_str, "status": "pending", "added_ts":
datetime.now(timezone.utc).isoformat()})
                updated_self = True
        if updated_self: self.log.info(f"Updated learning goals. Total: {len(self.learning_goals)}")

    if isinstance(reflection_data.get('adaptation_strategy_proposals'), list):
        for strat_str in reflection_data['adaptation_strategy_proposals']:
            # For simplicity, store as a list; more complex would parse key-value
            self.adaptation_strategies[f"proposal_{uuid.uuid4().hex[:8]}"] = strat_str
            updated_self = True
        if updated_self: self.log.info(f"Updated adaptation strategies. Total:
{len(self.adaptation_strategies)}")

    # (Other updates for knowledge_gaps, self_model_accuracy, etc. would follow similar
patterns)
    # Learned facts and prompt suggestions are now primarily handled by MemorySystem via
agent
    if reflection_data.get('learned_facts') or reflection_data.get('prompt_tuning_suggestions'):
        updated_kb_elements = True # Signal to agent to process these into MemorySystem
    if updated_self:
```

```
            self.log.info("SelfModel internal state updated from reflection.")
        return updated_self, updated_kb_elements

    def update_status(self, status: str):
        if status != self.current_status:
            self.log.debug(f"SelfModel status changing from '{self.current_status}' to '{status}'")
            self.current_status = status
            self.add_event_log(f"Status changed to {status}", event_type="status_update")

    def add_error_summary(self, error_info: Dict): # Takes dict now
        self.recent_errors.append(error_info)
        self.recent_errors = self.recent_errors[-MAX_RECENT_ERRORS_IN_STATE:]

    def backup_directives(self, reason: str):
        """Saves a backup of current directives to a file."""
        backup_file = SELF_MOD_BACKUP_DIR /
f"core_directives_backup_{datetime.now().strftime('%Y%m%d_%H%M%S')}_{reason}.json"
        try:
            with backup_file.open('w') as f:
                json.dump(self.core_directives, f, indent=2)
            self.log.info(f"Core directives backed up to {backup_file} due to: {reason}")
        except Exception as e:
            self.log.error(f"Failed to backup core directives: {e}")

    def simulate_internal_dialog(self, topic: str, perspectives: Optional[List[str]]=None) -> str:
        """Simulates an internal dialog about a topic using LLM, potentially from different
perspectives."""
        self.log.info(f"Simulating internal dialog on topic: {topic}")
        if perspectives is None:
            perspectives = ["analytical", "creative_explorer", "safety_officer"] # Default perspectives
        dialog_history = []
        full_dialog_str = f"Internal Dialog on: {topic}\n\n"
        for persp_idx, perspective_name in enumerate(perspectives):
            prompt = f"You are part of an AGI's internal dialog. Consider the topic: '{topic}'.\n"
            prompt += f"Adopt the perspective of a '{perspective_name}'. What are your thoughts,
questions, or suggestions?\n"
            if dialog_history:
                prompt += "\nPrevious contributions to this dialog:\n"
                for entry in dialog_history[-2:]: # Show last 2 contributions for context
                    prompt += f"- {entry['perspective']}: {entry['contribution']}\n"
            prompt += f"\nYour contribution (as {perspective_name}):"
            try:
                # Use _agent_instance_hack to access the LLM wrapper
                contribution = _agent_instance_hack.llm_wrapper.generate(prompt,
max_new_tokens=300, temperature=0.6) # type: ignore
                dialog_history.append({"perspective": perspective_name, "contribution":
contribution})
                full_dialog_str += f"Perspective ({perspective_name}): {contribution}\n\n"
            except Exception as e:
                self.log.error(f"Error during internal dialog generation for perspective
{perspective_name}: {e}")
                contribution = f"(Error generating contribution for {perspective_name})"
                full_dialog_str += f"Perspective ({perspective_name}): {contribution}\n\n"
```

```
        self.add_event_log(f"Internal dialog simulated on '{topic}'.", data={"dialog":
full_dialog_str})
        return full_dialog_str

class MotivationEngine:
    """Manages the agent's internal drives and their influence on behavior."""
    def __init__(self, drive_configs: Optional[Dict[DriveType, Dict[str, Any]]] = None):
        self.log = get_logger("MOTIVATION_ENGINE")
        self.drives: Dict[DriveType, DriveState] = {}
        self._initialize_drives(drive_configs)
        self.log.info("MotivationEngine initialized.")

    def _initialize_drives(self, drive_configs: Optional[Dict[DriveType, Dict[str, Any]]]):
        default_configs = {
            DriveType.CURIOSITY: {"decay_rate": 0.01, "max_level": 1.0, "min_level": 0.1,
"initial_level": 0.5},
            DriveType.MASTERY: {"decay_rate": 0.005, "max_level": 1.0, "min_level": 0.1,
"initial_level": 0.6},
            DriveType.ACHIEVEMENT: {"decay_rate": 0.015, "max_level": 1.0, "min_level": 0.1,
"initial_level": 0.5},
            DriveType.NOVELTY_SEEKING: {"decay_rate": 0.012, "max_level": 1.0, "min_level": 0.1,
"initial_level": 0.7},
            DriveType.PRESERVATION: {"decay_rate": 0.001, "max_level": 1.0, "min_level": 0.0,
"initial_level": 0.2}, # Health/Integrity
            DriveType.EFFICIENCY: {"decay_rate": 0.008, "max_level": 1.0, "min_level": 0.1,
"initial_level": 0.6},
            DriveType.SOCIAL_INTERACTION: {"decay_rate": 0.01, "max_level": 1.0, "min_level":
0.0, "initial_level": 0.3},
        }
        configs = drive_configs if drive_configs is not None else default_configs
        for drive_type in DriveType:
            config = configs.get(drive_type, default_configs.get(drive_type, {})) # type: ignore
            self.drives[drive_type] = DriveState(
                drive_type=drive_type,
                level=config.get("initial_level", 0.5),
                decay_rate=config.get("decay_rate", 0.01),
                max_level=config.get("max_level", 1.0),
                min_level=config.get("min_level", 0.0)
            )

    def update_drives(self):
        """Applies decay and updates drives based on recent experiences or time."""
        for drive_type, drive_state in self.drives.items():
            drive_state.update(stimulus=0.0) # Apply decay
            # More complex updates would happen here based on general perception/internal state

    def process_experience(self, experience: Experience):
        """Updates drives based on a specific experience."""
        # Example rules:
        if experience.type == "tool_output" and experience.content.get("success"):
            if experience.metadata.get("tool_name") == "learn":
                self.drives[DriveType.CURIOSITY].update(stimulus=-0.05) # Satiated
                self.drives[DriveType.MASTERY].update(stimulus=0.1) # Gained mastery
            elif "success" in experience.content.get("status", "").lower():
```

```
            self.drives[DriveType.ACHIEVEMENT].update(stimulus=0.05) # Achieved something
        elif experience.type == "error":
            self.drives[DriveType.PRESERVATION].update(stimulus=0.1) # Threat detected
            self.drives[DriveType.MASTERY].update(stimulus=-0.05) # Setback

    def get_drive_level(self, drive_type: DriveType) -> float:
        return self.drives.get(drive_type, DriveState(drive_type=drive_type)).level # Return default
if not found

    def get_all_drive_levels(self) -> Dict[DriveType, float]:
        return {dt: ds.level for dt, ds in self.drives.items()}

    def get_all_drive_levels_serializable(self) -> Dict[str, float]:
        return {dt.name: ds.level for dt, ds in self.drives.items()}

    def get_prioritized_drives(self, n: int = 3) -> List[Tuple[DriveType, float]]:
        """Returns top N drives by current level."""
        sorted_drives = sorted(self.drives.items(), key=lambda item: item[1].level, reverse=True)
        return [(dt, ds.level) for dt, ds in sorted_drives[:n]]

    def suggest_goal_type_from_drives(self) -> Optional[str]:
        """Suggests a goal type based on current highest drives."""
        top_drives = self.get_prioritized_drives(n=1)
        if not top_drives: return None

        top_drive_type = top_drives[0][0]
        if top_drive_type == DriveType.CURIOSITY:
            return "exploration"
        elif top_drive_type == DriveType.MASTERY:
            return "skill_improvement"
        elif top_drive_type == DriveType.ACHIEVEMENT:
            return "task_completion"
        elif top_drive_type == DriveType.PRESERVATION:
            return "self_maintenance"
        elif top_drive_type == DriveType.EFFICIENCY:
            return "optimization"
        return None

@dataclass
class DriveState:
    drive_type: DriveType
    level: float = 0.5
    decay_rate: float = 0.01 # Rate at which the drive naturally decreases
    max_level: float = 1.0
    min_level: float = 0.0
    last_update_time: float = field(default_factory=time.time)

    def update(self, stimulus: float = 0.0):
        """Updates the drive level based on stimulus and decay."""
        now = time.time()
        time_elapsed = now - self.last_update_time
        decay_amount = self.decay_rate * time_elapsed

        new_level = self.level - decay_amount + stimulus
```

```
        self.level = max(self.min_level, min(self.max_level, new_level))
        self.last_update_time = now

class FileChannel:
    """Implements a simple file-based communication channel for multi-agent systems."""
    def __init__(self, agent_id: str, shared_directory: str):
        self.agent_id = agent_id
        self.shared_dir = Path(shared_directory)
        self.shared_dir.mkdir(parents=True, exist_ok=True)
        self.inbox_file = self.shared_dir / f"inbox_{self.agent_id}.json"
        self.outbox_dir = self.shared_dir # Other agents read from here
        self.log = get_logger(f"COMMS_{agent_id}")
        self.handlers: Dict[MessageType, List[Callable[[Message], Optional[Message]]]] = {}
        self.log.info(f"FileChannel initialized for agent '{self.agent_id}'. Inbox: {self.inbox_file}")

    def _write_message_to_file(self, message: Message, target_file: Path) -> bool:
        try:
            # Use a file lock to prevent corruption during writes
            with FileLock(str(target_file) + ".lock", timeout=5): # type: ignore
                messages = []
                if target_file.exists():
                    try:
                        existing_content = target_file.read_text(encoding='utf-8')
                        if existing_content.strip(): # Avoid loading empty content as JSON
                            messages = json.loads(existing_content)
                    except json.JSONDecodeError as e:
                        self.log.error(f"Corrupted message file {target_file}: {e}. Clearing file.")
                        messages = [] # Reset if corrupted
                messages.append(message.to_dict())
                target_file.write_text(json.dumps(messages, indent=2), encoding='utf-8')
            return True
        except FileLockTimeout: # type: ignore
            self.log.warning(f"Timeout acquiring lock for {target_file}. Message not sent to file.")
            return False
        except Exception as e:
            self.log.error(f"Error writing message to {target_file}: {e}")
            return False

    def _read_messages_from_file(self, source_file: Path) -> List[Message]:
        messages = []
        if not source_file.exists():
            return []
        try:
            with FileLock(str(source_file) + ".lock", timeout=5): # type: ignore
                content = source_file.read_text(encoding='utf-8')
                if content.strip():
                    raw_messages = json.loads(content)
                    messages = [Message.from_dict(msg_data) for msg_data in raw_messages if
isinstance(msg_data, dict)]
                # Clear the file after reading
                source_file.write_text("", encoding='utf-8')
            return messages
        except FileLockTimeout: # type: ignore
            self.log.warning(f"Timeout acquiring lock for {source_file}. Cannot read messages.")
```

```
            return []
        except json.JSONDecodeError as e:
            self.log.error(f"Corrupted message file {source_file}: {e}. Clearing file.")
            source_file.write_text("", encoding='utf-8') # Clear corrupted file
            return []
        except Exception as e:
            self.log.error(f"Error reading messages from {source_file}: {e}")
            return []

    def send_message(self, message: Message) -> bool:
        target_inbox = self.shared_dir / f"inbox_{message.receiver_id}.json"
        message.sender_id = self.agent_id # Ensure sender is correct
        self.log.info(f"Sending {message.type} message to {message.receiver_id}:
{message.content.get('summary', str(message.content))[:50]}...")
        return self._write_message_to_file(message, target_inbox)

    def receive_messages(self) -> List[Message]:
        """Checks and retrieves new messages from the agent's inbox."""
        new_messages = self._read_messages_from_file(self.inbox_file)
        if new_messages:
            self.log.info(f"Received {len(new_messages)} new messages in inbox.")
        return new_messages

    def register_handler(self, message_type: MessageType, handler: Callable[[Message],
Optional[Message]]):
        """Registers a function to handle specific message types."""
        if message_type not in self.handlers:
            self.handlers[message_type] = []
        self.handlers[message_type].append(handler)
        self.log.debug(f"Registered handler for message type: {message_type.value}")

    def process_incoming_messages(self):
        """Processes all messages currently in the inbox using registered handlers."""
        messages = self.receive_messages() # This also clears the inbox file
        for msg in messages:
            self.log.debug(f"Processing message ID {msg.id}, Type: {msg.type}, From:
{msg.sender_id}")
            handled = False
            if msg.message_type in self.handlers:
                for handler_func in self.handlers[msg.message_type]:
                    try:
                        response = handler_func(msg)
                        if response:
                            self.send_message(response) # Send response if handler returns one
                        handled = True
                    except Exception as e:
                        self.log.error(f"Error handling message {msg.id} with handler
{handler_func.__name__}: {e}", exc_info=True)
                        # Optionally send an ERROR message back
                        self.send_message(Message(sender_id=self.agent_id,
receiver_id=msg.sender_id, type=MessageType.ERROR, content={"original_message_id":
msg.id, "error": str(e)}))
            if not handled:
```

```
            self.log.warning(f"No handler registered for message type
{msg.message_type.value}. Message ID {msg.id} unhandled.")


# --- Embodiment Abstraction Layer (Feature 7) ---
class Sensor(ABC):
    """Abstract base class for a sensor."""
    def __init__(self, id: str, embodiment: 'VirtualEmbodiment', config: Dict):
        self.id = id
        self.embodiment = embodiment
        self.config = config
        self.log = get_logger(f"SENSOR_{id}")

    @abstractmethod
    def get_reading(self) -> Any:
        """Returns the current reading from the sensor."""
        pass

class Actuator(ABC):
    """Abstract base class for an actuator."""
    def __init__(self, id: str, embodiment: 'VirtualEmbodiment', capabilities: List[str], config: Dict):
        self.id = id
        self.embodiment = embodiment
        self.capabilities = capabilities
        self.config = config
        self.log = get_logger(f"ACTUATOR_{id}")

    @abstractmethod
    def perform_action(self, action_type: str, **kwargs) -> Dict:
        """Performs a specific action using the actuator."""
        pass

class VirtualEmbodiment:
    """Simulated embodiment layer for AGI agents. (Can be replaced by Gym environments or
more complex sims)"""
    def __init__(self, agent: 'AutonomousAgent'):
        self.agent = agent
        self.log = get_logger("EMBODIMENT")
        self.location = "virtual_lab_control_room"
        self.state: Dict[str, Any] = {
            "health": 100, "energy": 100,
            "emotions": {"curiosity": 0.7, "focus": 0.8, "satisfaction": 0.5, "anxiety": 0.1},
            "internal_time": time.time(), # Simulated internal clock start
            "inventory": ["basic_manipulator_tool", "data_logger_module"],
            "active_sensors": ["text_interface", "internal_state_monitor"],
            "world_model_accuracy": 0.6 # Agent's own estimate
        }
        self.environment_map = self._init_env()
        self.sensors: Dict[str, Sensor] = {}
        self.actuators: Dict[str, Actuator] = {}
        self.sensory_log: List[Dict] = []
        self.MAX_SENSORY_LOG_SIZE = 100
        # Gym environment (optional)
        self.gym_env = None
```

```
        # if GYMNASIUM_AVAILABLE:
        #     try:
        #         self.gym_env = gym.make("CartPole-v1") # Example environment
        #         self.log.info("Gymnasium environment 'CartPole-v1' loaded.")
        #     except Exception as e:
        #         self.log.warning(f"Could not load example Gym environment: {e}")

    def _init_env(self) -> Dict[str, Any]:
        return {
            "virtual_lab_control_room": {
                "description": "A brightly lit control room with multiple holographic displays showing
system diagnostics. A console provides interaction with the core AGI systems. Doors lead to
'Data Center' and 'Simulation Bay'.",
                "objects": ["diagnostic_console", "emergency_shutdown_button",
"research_terminal"],
                "exits": {"north": "simulation_bay", "east": "data_center"},
                "features": ["interactive_console"]
            },
            "simulation_bay": {
                "description": "A large, reconfigurable bay designed for running complex simulations.
Currently, a simple robotics arm simulation is active on one of the platforms.",
                "objects": ["robotics_arm_simulation_interface", "environment_config_panel"],
                "exits": {"south": "virtual_lab_control_room"},
                "features": ["simulation_runner"]
            },
            "data_center": {
                "description": "Rows of servers hum quietly. Access panels show data flow and
storage capacity.",
                "objects": ["main_database_interface", "backup_power_control"],
                "exits": {"west": "virtual_lab_control_room"},
                "features": ["data_management_interface"]
            },
            "agi_core_chamber": { # Added for conceptual interaction
                "description": "A shielded chamber housing the agent's primary cognitive core.
Direct interaction is limited for safety.",
                "objects": ["core_status_monitor", "directive_override_terminal_SECURE"],
                "exits": {}, # No easy exits, conceptual space
                "features": ["introspection_interface"]
            }
        }

    def add_sensor(self, sensor: Sensor):
        self.sensors[sensor.id] = sensor
        self.log.info(f"Added sensor: {sensor.id}")

    def add_actuator(self, actuator: Actuator):
        self.actuators[actuator.id] = actuator
        self.log.info(f"Added actuator: {actuator.id}")

    def list_sensors(self) -> List[Dict]:
        return [{"id": s.id, "type": s.__class__.__name__} for s in self.sensors.values()]

    def list_actuators(self) -> List[Dict]:
```

```
        return [{"id": a.id, "type": a.__class__.__name__, "capabilities": a.capabilities} for a in
self.actuators.values()]

    def get_sensory_input(self) -> List[Dict[str, Any]]:
        """Generate synthetic sensory input based on current environment and internal state."""
        self.log.debug(f"Embodiment generating sensory input. Location: {self.location}")
        env_details = self.environment_map.get(self.location, {})
        # Simulate passage of time for internal state
        self.state["internal_time"] = time.time()
        self.state["energy"] = max(0, self.state["energy"] - 0.1) # Slow energy decay
        if self.state["energy"] < 20: self.state["emotions"]["anxiety"] = min(1.0,
self.state["emotions"].get("anxiety",0) + 0.1)

        sensory_packet = {
            "timestamp": datetime.now(timezone.utc).isoformat(),
            "type": "environment_scan",
            "source": "virtual_embodiment",
            "content": {
                "location": self.location,
                "description": env_details.get("description", "An undefined space."),
                "visible_objects": env_details.get("objects", []),
                "available_exits": list(env_details.get("exits", {}).keys()),
                "special_features": env_details.get("features", [])
            }
        }
        internal_state_packet = {
            "timestamp": datetime.now(timezone.utc).isoformat(),
            "type": "internal_state_report",
            "source": "virtual_embodiment_self_monitor",
            "content": copy.deepcopy(self.state) # Report a copy of internal state
        }
        self.sensory_log.append(sensory_packet)
        self.sensory_log.append(internal_state_packet)
        if len(self.sensory_log) > self.MAX_SENSORY_LOG_SIZE:
            self.sensory_log = self.sensory_log[-self.MAX_SENSORY_LOG_SIZE:]

        # Conceptual: If Gym environment is active, get observation from it
        # gym_obs_packet = None
        # if self.gym_env:
        #     try:
        #         # This needs proper state management for the gym env (reset, step)
        #         # For now, just a conceptual placeholder of getting an observation
        #         # gym_observation, reward, terminated, truncated, info =
self.gym_env.step(self.gym_env.action_space.sample()) # Example action
        #         # gym_observation = self.gym_env.observation_space.sample() # Get a sample
observation
        #         # gym_obs_packet = {
        #         #     "timestamp": datetime.now(timezone.utc).isoformat(),
        #         #     "type": "gym_observation",
        #         #     "source": self.gym_env.spec.id if self.gym_env.spec else "gym_env",
        #         #     "content": {"observation": str(gym_observation)} # Convert to string for simple
logging
        #         # }
        #         # self.sensory_log.append(gym_obs_packet)
```

```
        #     except Exception as e:
        #         self.log.error(f"Error interacting with Gym environment: {e}")
        return [sensory_packet, internal_state_packet] # Could add gym_obs_packet if not None

    def act(self, action_type: str, target: Optional[str] = None, params: Optional[Dict] = None) ->
Dict:
        """
        Simulates the agent performing an action in the virtual world.
        Returns a dictionary with the result of the action.
        """
        self.log.info(f"Embodiment performing action: {action_type}, Target: {target}, Params:
{params}")
        params = params or {}
        env_details = self.environment_map.get(self.location, {})
        result_status = "failure"
        message = f"Action '{action_type}' on '{target}' could not be performed as specified."

        # Basic world interactions
        if action_type == "move":
            if target and target in env_details.get("exits", {}):
                new_location = env_details["exits"][target]
                self.location = new_location
                self.state["emotions"]["curiosity"] = min(1.0, self.state["emotions"]["curiosity"] + 0.1)
                message = f"Moved to {new_location}."
                result_status = "success"
            else:
                message = f"Cannot move to '{target}' from {self.location}."
        elif action_type == "examine":
            if target and target in env_details.get("objects", []):
                message = f"You examine the {target}. It appears to be a standard {target}."
                if target == "diagnostic_console": message += " It shows fluctuating green and
amber lights."
                elif target == "core_status_monitor": message += " It indicates: Core Nominal.
Directives Stable. Learning Rate: Optimal."
                result_status = "success"
            elif target and target in env_details.get("features", []):
                message = f"You examine the feature: {target}. It seems operational."
                result_status = "success"
            else:
                message = f"There is no '{target}' to examine here."
        elif action_type == "pickup": # Simplified
            if target and target in env_details.get("objects", []):
                env_details["objects"].remove(target) # type: ignore
                self.state["inventory"].append(target)
                self.state["emotions"]["satisfaction"] = min(1.0, self.state["emotions"]["satisfaction"] +
0.2)
                message = f"Picked up {target}."
                result_status = "success"
            else:
                message = f"Cannot pickup '{target}'."
        elif action_type == "use_feature": # New action type
            feature_name = target
            if feature_name and feature_name in env_details.get("features",[]):
                if feature_name == "interactive_console":
```

```
                # Conceptual: What does interacting with the console do?
                # This could involve presenting the agent with more detailed info or a sub-problem.
                console_output = self._use_interactive_console(params.get("command"))
                message = f"Interacted with {feature_name}. Output: {console_output}"
                result_status = "success"
            elif feature_name == "simulation_runner" and self.location == "simulation_bay":
                sim_result = self._run_simulation(params.get("simulation_name", "default_physics_test"), params.get("config",{}))
                message = f"Ran simulation '{params.get('simulation_name')}'. Result: {sim_result}"
                result_status = "success"
            else:
                message = f"Feature '{feature_name}' used. Generic interaction occurred."
                result_status = "success" # Assume simple use is always successful for now
        else:
            message = f"No feature '{feature_name}' to use here."
    elif action_type == "rest":
        self.state["energy"] = min(100, self.state["energy"] + random.randint(10, 20))
        self.state["emotions"]["anxiety"] = max(0.0, self.state["emotions"]["anxiety"] - 0.2)
        self.state["emotions"]["satisfaction"] = min(1.0, self.state["emotions"]["satisfaction"] + 0.1)
        message = "You rest and regain some energy. Anxiety decreases."
        result_status = "success"

    # Affect emotional state based on action outcome
    self._modulate_emotions(action_type, result_status)

    return {"status": result_status, "message": message, "new_location": self.location if action_type=="move" else None, "updated_inventory": self.state["inventory"] if action_type=="pickup" else None}

def _modulate_emotions(self, action: str, status: str):
    em = self.state["emotions"]
    if status == "success":
        em["satisfaction"] = min(1.0, em["satisfaction"] + 0.05)
        em["anxiety"] = max(0.0, em["anxiety"] - 0.02)
        if action in ["explore", "examine", "move_to_new_area"]:
            em["curiosity"] = min(1.0, em["curiosity"] + 0.1)
    elif status == "failure":
        em["satisfaction"] = max(0.0, em["satisfaction"] - 0.1)
        em["anxiety"] = min(1.0, em["anxiety"] + 0.05)
        # Frustration could be a derived emotion: high anxiety + low satisfaction

    # Decay curiosity if not exploring
    if action not in ["explore", "examine", "move_to_new_area", "learn"]:
        em["curiosity"] = max(0.1, em["curiosity"] - 0.01) # Maintain a base level
    for key in em: # Clamp values
        em[key] = round(min(1.0, max(0.0, em[key])), 2)
    self.log.debug(f"Emotions modulated: {em}")


def _use_interactive_console(self, command: Optional[str]) -> str:
    if command:
```

```
            self.log.info(f"Embodiment: Console command received: '{command}'")
            if "diagnostics" in command.lower():
                return "System Diagnostics: All core modules report nominal status. Memory usage
at 65%. CPU load at 30%."
            elif "query_self_model" in command.lower():
                return f"Self-Model Query Response:
{self.agent.self_model.internal_state_narrative[:100]}..."
            else:
                return f"Console command '{command}' executed. (Mock Response)"
        return "Console ready for input. Available commands: 'diagnostics', 'query_self_model
<topic>'."

    def _run_simulation(self, sim_name: str, config: Dict) -> str:
        self.log.info(f"Embodiment: Running simulation '{sim_name}' with config: {config}")
        # Placeholder for actual simulation logic
        # This could interact with a Gym environment or a more complex simulator.
        time.sleep(random.uniform(0.5, 2.0)) # Simulate time taken
        success_chance = 0.8
        if "risky_config" in config: success_chance = 0.4
        if random.random() < success_chance:
            outcome_value = random.randint(50,100)
            self.state["emotions"]["satisfaction"] = min(1.0, self.state["emotions"]["satisfaction"] +
0.3)
            self.state["emotions"]["curiosity"] = min(1.0, self.state["emotions"]["curiosity"] + 0.1)
            # Conceptual: agent learns from simulation outcome
            # self.agent.learning_module.add_experience(...)
            return f"Simulation '{sim_name}' completed successfully. Outcome metric:
{outcome_value}."
        else:
            error_code = random.randint(1000,2000)
            self.state["emotions"]["anxiety"] = min(1.0, self.state["emotions"]["anxiety"] + 0.2)
            return f"Simulation '{sim_name}' failed. Error code: {error_code}. Check configuration."

    def summary(self) -> str:
        """Returns a string summary of the embodiment's current state."""
        return (
            f"Location: {self.location}\n"
            f"Description: {self.environment_map.get(self.location, {}).get('description')}\n"
            f"Visible Objects: {self.environment_map.get(self.location, {}).get('objects')}\n"
            f"Exits: {list(self.environment_map.get(self.location, {}).get('exits', {}).keys())}\n"
            f"Internal State (summary): Energy={self.state['energy']},
Emotions={self.state['emotions']}, Inventory={self.state['inventory']}"
        )


# --- CognitiveCycle (AGI Enhancements) ---
class CognitiveCycle:
    def __init__(self, agent: 'AutonomousAgent'):
        self.agent = agent
        self.log = get_logger("COGNITIVE_CYCLE")
        self.last_perception_time: float = 0.0
        # Initialize AGI modules used within the cycle
        self.perception_module = PerceptionModule(agent)
        self.planning_module = PlanningModule(agent)
```

```
        # Understanding and Deliberation are more deeply integrated into perceive/deliberate
methods

    def run_cycle(self) -> bool:
        global LAST_METACOGNITIVE_CHECK_CYCLE,
LAST_LEARNING_MODULE_UPDATE_CYCLE
        self.log.debug(f"--- Starting Cognitive Cycle {self.agent.cycle_count} --- Status:
{self.agent.self_model.current_status if self.agent.self_model else 'N/A_SM'}, Goal Stack
Depth: {len(self.agent.goal_stack)}")

        self.agent.current_goal_outcome = None # Reset for this cycle
        active_goal_before_cycle_dict = copy.deepcopy(self.agent.state['goals'].get('active')) if
self.agent.state['goals'].get('active') else None
        self.agent.last_error = None
        self.agent.current_goal_outcome = None # Reset for this cycle
        cycle_ok = False
        try:
            # 1. Perception
            observations = self.perception_module.perceive()
            self.last_perception_time = time.time()
            # 2. Understanding (Integrate raw observations into a coherent world model update)
            understanding_result = self._understand(observations) # Returns structured
understanding

            # AGI: Proactive Metacognitive Check (as in OCR)
            if self.agent.self_model and (self.agent.cycle_count -
LAST_METACOGNITIVE_CHECK_CYCLE >= METACOGNITIVE_CHECK_INTERVAL_CYCLES):
                self.log.info(f"Triggering proactive metacognitive check (Cycle
{self.agent.cycle_count}).")
                anomalies = self.agent.self_model.perform_metacognitive_check()
                LAST_METACOGNITIVE_CHECK_CYCLE = self.agent.cycle_count
                if anomalies:
                    self.log.warning(f"Metacognitive anomalies detected: {anomalies}")
                    for anomaly_desc in anomalies:
                        self.agent._create_metacognitive_goal(anomaly_desc) # Creates a high-priority
goal

            # AGI: Trigger Learning Module periodically or based on events
            if self.agent.learning_module and (self.agent.cycle_count -
LAST_LEARNING_MODULE_UPDATE_CYCLE >=
LEARNING_MODULE_UPDATE_INTERVAL_CYCLES):
                self.log.info(f"Triggering learning module update (Cycle {self.agent.cycle_count}).")
                self.agent.learning_module.learn_from_recent_experiences()
                LAST_LEARNING_MODULE_UPDATE_CYCLE = self.agent.cycle_count

            # 3. Deliberation (Goal management, selection, or generation)
            # Deliberation now produces chosen_action_type, next_goal_to_execute (full Goal dict),
new_pending_goals
            deliberation_decision = self._deliberate(understanding_result)
            action_type = deliberation_decision.get("chosen_action_type", "idle")
            next_goal_dict_from_delib = deliberation_decision.get("next_goal") # This is a Goal dict
            newly_generated_pending_goals = deliberation_decision.get("new_pending_goals", []) #
List of Goal dicts
```

```
        # Add any newly generated pending goals from deliberation
        if newly_generated_pending_goals:
            with self.agent.lock:
                pending_list = self.agent.state['goals'].setdefault('pending', [])
                for ng_dict in newly_generated_pending_goals:
                    if isinstance(ng_dict, dict): # Should be a Goal dict
                        # Minimal validation or conversion if needed
                        # For now, assume it's a valid Goal dict ready to be stored
                        if not any(p['id'] == ng_dict['id'] for p in pending_list): # Avoid duplicates
                            pending_list.append(ng_dict)
                        else:
                            self.log.debug(f"Skipping duplicate pending goal from deliberation:
{ng_dict.get('id')}")
                    else:
                        self.log.warning(f"Deliberation produced non-dict pending goal: {ng_dict}")
                pending_list.sort(key=lambda x: GoalPriority[x.get('priority', 'MEDIUM').upper() if
isinstance(x.get('priority'),str) else GoalPriority(x.get('priority',
GoalPriority.MEDIUM)).name ].value, reverse=True) # type: ignore
                self.agent.save_state() # Save new pending goals

        goal_to_execute_this_cycle_dict: Optional[Dict] = None
        if action_type == "new_goal" or action_type == "pending_goal" or action_type ==
"active_goal_continue": # A goal was selected/activated by deliberation
            if next_goal_dict_from_delib and isinstance(next_goal_dict_from_delib, dict):
                goal_to_execute_this_cycle_dict = next_goal_dict_from_delib
                # Agent's main state 'active' goal is set by _deliberate
                self.log.info(f"Deliberation selected goal for execution:
{goal_to_execute_this_cycle_dict.get('goal', 'N/A')[:50]} (ID:
{goal_to_execute_this_cycle_dict.get('id')})")
            else: # Fallback if deliberation chose goal but didn't provide it
                self.log.warning(f"Deliberation chose '{action_type}' but no valid 'next_goal'
provided. Idling.")
                action_type = "idle"

        # 4. Plan & 5. Act (if a goal is set for this cycle)
        if goal_to_execute_this_cycle_dict:
            current_goal_obj = Goal.from_dict(goal_to_execute_this_cycle_dict) # Work with Goal
object

            # Check if plan already exists and is valid, or generate/re-plan
            # This simplified logic assumes plan is a list of steps in the goal dict.
            # A more robust system would check plan validity against current world state.
            if not current_goal_obj.plan or current_goal_obj.status == GoalStatus.PENDING or
current_goal_obj.replan_count > 0: # PENDING implies new or needs fresh plan
                # If replan_count > 0, it means a previous plan failed and needs new plan or re-
planning
                plan_steps, thought_str = self.planning_module.generate_plan(current_goal_obj)
                current_goal_obj.plan = plan_steps
                current_goal_obj.thought = thought_str
                current_goal_obj.replan_count = 0 # Reset replan_count after successful plan
generation
                # Update the active goal in agent state with the new plan
                with self.agent.lock:
                    self.agent.state['goals']['active'] = current_goal_obj.to_dict()
```

```
                    self.agent.save_state() # Save new plan

            if current_goal_obj.plan: # If plan exists (or was just generated)
                self.agent.current_goal_outcome = self._act(current_goal_obj) # _act now takes
Goal object
            else: # Planning failed or no plan
                self.log.warning(f"Plan available or generated for goal: {current_goal_obj.goal[:50]}.
Goal may fail.")
                self.agent.current_goal_outcome = False # Treat as failure
        else: # No specific goal, agent is idle or performing non-goal action
            self.agent.current_goal_outcome = True # Idle is a "successful" cycle in a way
            # Perform idle deliberation if configured
            if time.time() - LAST_DELIBERATION_TIME >
IDLE_DELIBERATION_INTERVAL_SECONDS:
                self.log.info("Performing idle deliberation...")
                # This could involve reviewing pending goals, self-improvement tasks, exploration
                # For now, just log and reset timer.
                # self._perform_idle_deliberation_tasks() # A new method could handle this
                global LAST_DELIBERATION_TIME
                LAST_DELIBERATION_TIME = time.time() # type: ignore

        cycle_ok = True # Cycle completed successfully (even if goal failed or agent was idle)

    except (PlanningError, ExecutionError, ToolNotFoundError, CodeGenerationError,
            SelfModificationError, LogicError, LLMError, SecurityError, ConfigurationError,
            MemoryError, PerceptionError, UnderstandingError, DeliberationError,
            RecursionDepthError, SimulationError, CommunicationError, EmbodimentError,
            LearningError, SafetyViolationError) as agent_cycle_err:
        # These are "controlled" errors expected within a cycle related to a specific goal
attempt.
        self.log.error(f"Cognitive cycle terminated for current goal processing due to Agent
Error: {agent_cycle_err}", exc_info=False)
        self.agent.current_goal_outcome = False
        self.agent.last_error = agent_cycle_err
        cycle_ok = True # Cycle finished (with an error for current goal), but agent can continue
unless critical.
    except Exception as critical_err:
        # Catch truly unexpected critical errors within the cycle's main try block.
        self.log.critical(f"CRITICAL Cognitive Cycle Error: {critical_err}", exc_info=True)
        self.agent.current_goal_outcome = False
        self.agent.last_error = critical_err
        STOP_SIGNAL_RECEIVED.set() # Critical failure, signal agent shutdown
        cycle_ok = False

    finally:
        self.log.debug(f"--- Cognitive Cycle {self.agent.cycle_count} Finished ({time.time() -
start_time:.3f}s) ---")
        # The active_goal_data_before_cycle (goal active at START of cycle)
        # is archived in the main agent loop using self.agent.current_goal_outcome.
    return cycle_ok

@retry(attempts=2, delay=2, retry_on=(LLMError, UnderstandingError))
def _understand(self, observations: List[Dict]) -> Dict[str, Any]:
    """Processes observations to update world model and identify key information."""
```

```
        self.log.debug(f"Understanding {len(observations)} observations...")
        understanding_summary = "Observations processed."
        processed_info = {"relevant_entities": [], "key_events": [], "state_changes": []}
        # Example: Use LLM to summarize and extract key info from observations
        # This is a simplified approach. A real system might have more structured parsing.
        if observations:
            prompt = "You are an AI agent. Synthesize the following observations into a coherent
understanding of the current situation. Identify key entities, events, and any significant changes
in the environment or your internal state. Focus on information relevant to achieving current
goals.\n\nObservations:\n"
            for obs in observations:
                prompt += f"- Type: {obs.get('type')}, Source: {obs.get('source')}, Content:
{str(obs.get('content'))[:200]}...\n" # Limit content length in prompt
            prompt += "\nProvide your synthesis as a JSON object: {\"summary_of_situation\":
\"str\", \"key_entities_mentioned\": [\"str\"], \"notable_events_or_changes\": [\"str\"],
\"potential_impact_on_goals\": \"str\"}"
            try:
                llm_response_str = self.agent.llm_wrapper.generate(prompt, max_new_tokens=500)
                synthesis = extract_json_robust(llm_response_str)
                if not synthesis.get("error"):
                    understanding_summary = synthesis.get("summary_of_situation",
understanding_summary)
                    processed_info["relevant_entities"] = synthesis.get("key_entities_mentioned", [])
                    processed_info["key_events"] = synthesis.get("notable_events_or_changes", [])
                    processed_info["potential_impact"] = synthesis.get("potential_impact_on_goals")
                    self.log.info(f"Understanding synthesized: {understanding_summary[:100]}...")
                    # Update MemorySystem with new structured understanding or raw observations
as facts/experiences
                    for event_str in processed_info["key_events"]:
                        exp_entry = Experience(content=event_str, type="environment_event",
metadata={"source": "perception_synthesis"})
                        self.agent.memory_system.add_memory_entry(exp_entry,
persist_to_vector=True)
                else:
                    self.log.warning(f"LLM failed to synthesize understanding: {synthesis.get('error')}")
            except Exception as e:
                self.log.error(f"Error during LLM-based understanding: {e}")
        # Update agent's internal world model representation (conceptual)
        # self.agent.memory_system.update_world_model(processed_info) # This would involve
complex updates to the graph/relational store
        return {"summary": understanding_summary, "processed_info": processed_info,
"raw_observations": observations}

    @retry(attempts=MAX_REPLAN_ATTEMPTS, delay=3, retry_on=(LLMError,
DeliberationError)) # Uses config
    def _deliberate(self, understanding_result: Dict) -> Dict:
        """
        Core deliberation logic: goal management, selection, and generation.
        Returns a dict: {"chosen_action_type": str, "next_goal": Optional[Goal_dict],
                         "new_pending_goals": List[Goal_dict]}
        """
        global LAST_DELIBERATION_TIME
        LAST_DELIBERATION_TIME = time.time()
        self.log.info("Deliberating on current situation and goals...")
```

```
        # Default action is to idle if no goals are pressing
        decision = {"chosen_action_type": "idle", "next_goal": None, "new_pending_goals": []}

        # Review pending goals
        pending_goals = self.agent.state['goals'].get('pending', [])
        active_goal_dict = self.agent.state['goals'].get('active') # Current active goal
        # Sort pending goals by priority (descending)
        # Ensure GoalPriority objects are used for sorting if not already converted
        def get_priority_val(goal_dict_item):
            p = goal_dict_item.get('priority', GoalPriority.MEDIUM.value)
            if isinstance(p, GoalPriority): return p.value
            if isinstance(p, str): return GoalPriority[p.upper()].value
            return p # Assume it's already int value if not str/enum
        pending_goals.sort(key=get_priority_val, reverse=True)
        highest_priority_pending: Optional[Dict] = None
        if pending_goals:
            highest_priority_pending = pending_goals[0]

        # Preemption logic: Can a pending goal preempt the active one?
        if active_goal_dict and highest_priority_pending:
            active_priority_val = get_priority_val(active_goal_dict)
            pending_priority_val = get_priority_val(highest_priority_pending)
            if pending_priority_val > active_priority_val: # Higher number = higher priority
                self.log.info(f"Pending goal '{highest_priority_pending.get('goal')}' (Prio:
{pending_priority_val}) preempts active goal '{active_goal_dict.get('goal')}' (Prio:
{active_priority_val}).")

                # Pause current active goal and push to stack
                with self.agent.lock:
                    paused_goal_dict = copy.deepcopy(active_goal_dict)
                    paused_goal_dict['status'] = GoalStatus.PAUSED.value
                    self.agent.goal_stack.append({'goal_data': paused_goal_dict, 'snapshot_time':
datetime.now(timezone.utc).isoformat()})
                    self.log.info(f"Pushed active goal '{active_goal_dict.get('goal')[:30]}' to stack
(paused).")

                # Activate the new highest priority pending goal
                new_active_goal_dict = pending_goals.pop(0) # Remove from pending
                new_active_goal_dict['status'] = GoalStatus.ACTIVE.value
                self.agent.state['goals']['active'] = new_active_goal_dict
                decision["chosen_action_type"] = "pending_goal" # Indicates a pending goal was
activated
                decision["next_goal"] = new_active_goal_dict
                self.agent.save_state()
                return decision

        # If there's an active goal and it wasn't preempted, continue it
        if active_goal_dict:
            self.log.debug(f"Continuing with active goal: {active_goal_dict.get('goal')[:50]}")
            decision["chosen_action_type"] = "active_goal_continue" # Special type to indicate
continue
            decision["next_goal"] = active_goal_dict
            return decision
```

```
        # If no active goal, and there are pending goals, activate the highest priority one
        if not active_goal_dict and highest_priority_pending:
            with self.agent.lock:
                new_active_goal_dict = pending_goals.pop(0) # Remove from pending
                new_active_goal_dict['status'] = GoalStatus.ACTIVE.value
                self.agent.state['goals']['active'] = new_active_goal_dict
                decision["chosen_action_type"] = "pending_goal"
                decision["next_goal"] = new_active_goal_dict
                self.agent.save_state()
                return decision

        # If no active or pending goals, consider generating a new one based on understanding or
directives (idle task)
        if not active_goal_dict and not pending_goals:
            self.log.info("No active or pending goals. Considering idle tasks or new goal
generation.")
            # Conceptual: LLM call to suggest an idle task or a new goal based on situation/
directives
            # Example: Create a default learning/exploration goal if truly idle
            if time.time() - LAST_DELIBERATION_TIME >=
IDLE_DELIBERATION_INTERVAL_SECONDS:
                idle_goal = Goal(
                    goal="Perform general self-assessment and explore the virtual environment.",
                    priority=GoalPriority.LOW,
                    origin="idle_deliberation",
                    associated_directive_ids=["directive_learn", "directive_curiosity",
"directive_metacog"]
                ).to_dict()
                decision["new_pending_goals"].append(idle_goal)
                decision["chosen_action_type"] = "idle_new_goal_generated" # if a new goal is made
for idle time
                pass # Continue to prompt LLM for deliberation

            # Check for sub-goal requests from tool executions (now handled by adding to
new_pending_goals)
            # If a tool like `execute_sub_goal` was called, its result (`sub_goal_data`) should be
processed
            # by the agent loop and might be part of `understanding_result` or a special flag.
            # Let's assume `understanding_result['raw_observations']` might contain such signals.
            for obs in understanding_result.get('raw_observations', []):
                if obs.get('type') == 'tool_result' and obs.get('content',{}).get('status') ==
'sub_goal_prepared':
                    sub_goal_dict = obs['content'].get('sub_goal_data')
                    if sub_goal_dict and isinstance(sub_goal_dict, dict):
                        self.log.info(f"Deliberation found sub-goal request from tool output:
{sub_goal_dict.get('goal')[:50]}")
                        # Ensure it's not already pending to avoid duplicates
                        if not any(p['id'] == sub_goal_dict['id'] for p in decision["new_pending_goals"]):
                            decision["new_pending_goals"].append(sub_goal_dict)
                        # Immediately activate it if no current active goal
                        if not active_goal_dict:
                            self.log.info(f"Activating immediate sub-goal: {sub_goal_dict.get('goal')[:50]}")
                            sub_goal_dict['status'] = GoalStatus.ACTIVE.value
                            self.agent.state['goals']['active'] = sub_goal_dict
```

```
                    decision["chosen_action_type"] = "new_goal"
                    decision["next_goal"] = sub_goal_dict
                    with self.agent.lock:
                        self.agent.save_state()
                    return decision # Return early to prioritize this immediate sub-goal

        # If still no specific action, LLM will decide
        self_model_summary =
self.agent.self_model.get_summary_for_prompt(include_tool_reliability=True)
        understanding_summary = understanding_result.get('summary', 'No specific
understanding summary.')
        pressing_issue = understanding_result.get('processed_info',{}).get('potential_impact',
'None identified.') # Use potential_impact
        interp_con_val = understanding_result.get('interpretation_confidence', 0.7)
        recent_memory_context =
self.agent.memory_system.get_knowledge_summary_for_prompt(understanding_summary,
max_facts=3) # type: ignore

        prompt_parts = [
            f"**Deliberation Context for {AGENT_NAME}:**",
            f"* **Self-Model Snapshot:**\n{self_model_summary}",
            f"* **Current Understanding (Confidence: {interp_con_val:.2f}):**
{understanding_summary}",
            f"* **Most Pressing Issue/Opportunity Identified:** {pressing_issue}",
            f"* **Recent Key Memories:**\n{recent_memory_context}",
            f"* **Short-Term Memory
(STM):**\n{self.agent.memory_system.get_short_term_memory_summary()}",
            f"* **Pending Goals ({len(pending_goals)}):** {json.dumps([g for g in pending_goals[:3]],
indent=2)}",
            f"* **Current Active Goal:** {'None' if not active_goal_dict else
f'{active_goal_dict.get("goal")[:100]}... (ID: {active_goal_dict.get("id")})'}",
            f"* **Agent Core Directives
(Weighted):**\n{json.dumps(self.agent.self_model.core_directives, indent=2)}\n",
            "**Task: Advanced Deliberation & Action Selection**",
            "1. **Analyze Situation & Drives:** Based on ALL context (self-model, understanding,
drives, memories, goals, directives), what is the most critical aspect demanding attention or the
best opportunity for progress? Current Drives (Scale 0-1, High=Strong): " + \
            f"{self.agent.self_model.motivation_engine.get_all_drive_levels_serializable()}.",
            "2. **Generate Options:** Propose potential actions or new goals. Consider:",
            "    - Responding to user commands (if any, as 'user_command' type in observations).",
            "    - Continuing current `active_goal` (if suitable and has a plan).",
            "    - Selecting the highest priority `pending_goal` (if `active_goal` is unsuitable/
complete).",
            "    - Performing `reflection` or `self_assessment` (if mandatory timers, drives like low
CONFIDENCE, or pressing issues suggest it).",
            "    - Generating `new_goal`(s) based on Drives (e.g., high CURIOSITY -> exploration
goal), Directives (e.g., low-eval directive -> improvement goal), or identified opportunities. New
goals require `goal` (str), `priority` (float 0.0-1.0), `origin` (str e.g., 'drive_curiosity',
'directive_alignment'). Optional: `context_for_planning` (dict), `associated_directive_ids` (list of
str).",
            "    - (Conceptual) Simulate 1-2 high-priority new goal ideas or current plan steps for
viability before committing if uncertainty is high or consequence severe (briefly note simulation
outcome).",
```

```
            "    - Remaining `idle` if no pressing tasks and no valuable proactive actions are apparent. Use `idle_new_goal_generated` if you create a new goal as part of being idle.",
            "3. **Prioritize & Select:** Choose the SINGLE most appropriate action/goal for the *immediate next cycle*. Justify your choice, especially if it deviates from obvious triggers, high drives, or highest priority pending. State reasoning clearly.",
            ```python
            "4. **Manage Goal List:** If generating new goals, add them to `new_pending_goals` list. If selecting an existing pending goal, it moves to `next_goal` and is removed from pending internally (do not include in `new_pending_goals` output).",
            "5. **Output ONLY a JSON object with the following keys:**",
            "    - `reasoning`: (string) Your detailed thought process for the decision, including drive/directive considerations and option evaluation.",
            "    - `chosen_action_type`: (string) One of: 'resume_active_goal', 'pending_goal', 'new_goal', 'reflection', 'self_assessment', 'external_command_action', 'idle', 'idle_new_goal_generated'.",
            "    - `next_goal`: (object:Goal or null) The *full goal object* (matching Goal dataclass structure) selected for immediate execution. Null if idle/reflection/assessment without a direct goal target.",
            "    - `new_pending_goals`: (list of object:Goal) Any *newly generated* goals (not chosen for immediate execution). Include full Goal objects. Empty list if no new goals generated.",
            "CRITICAL: Do NOT put an already existing pending goal that you selected into `new_pending_goals`. `next_goal` handles that. Only truly NEWLY conceptualized goals go into `new_pending_goals`."
        ]
        deliberation_prompt = "\n".join(prompt_parts)
        self.log.debug(f"Deliberation prompt for LLM: \n{deliberation_prompt}")

        if not self.agent.llm_wrapper:
            raise LLMError("LLMWrapper not available for deliberation.")

        deliberation_llm_response = self.agent.llm_wrapper.generate(
            deliberation_prompt,
            system_message="You are the core deliberation faculty of an advanced AI agent. Analyze the situation comprehensively, consider drives and directives, and make strategic decisions. Respond ONLY in JSON as per output instructions.",
            temperature=0.5 # Balance creativity and consistency for deliberation
        )

        # Ensure error handling as in other LLM calls...
        if extract_json_robust(deliberation_llm_response).get("error") and not \
           (isinstance(deliberation_llm_response,str) and deliberation_llm_response.strip().startswith("{")):
            raise DeliberationError(f"LLM deliberation call failed or returned non-JSON: {extract_json_robust(deliberation_llm_response).get('error')}")

        deliberation_decision = extract_json_robust(deliberation_llm_response)

        # Validate structure
        required_delib_keys = ['reasoning', 'chosen_action_type', 'next_goal', 'new_pending_goals']
        for key in required_delib_keys:
            if key not in deliberation_decision:
                self.log.error(f"Deliberation JSON response missing key: '{key}'. Received keys: {deliberation_decision.keys()}")
```
```

```
            # Default based on key type to prevent crashes
            if key == 'new_pending_goals': deliberation_decision[key] = []
            elif key == 'next_goal': deliberation_decision[key] = None
            else: deliberation_decision[key] = "Error: Missing from LLM Output"

        # Validate types further
        if not isinstance(deliberation_decision.get('new_pending_goals'), list):
            self.log.warning("Deliberation 'new_pending_goals' was not a list. Resetting to empty
list.")
            deliberation_decision['new_pending_goals'] = []

        if deliberation_decision.get('next_goal') is not None and not
isinstance(deliberation_decision.get('next_goal'), dict):
            self.log.warning(f"Deliberation 'next_goal' was not a dict or null. Setting to null. Value:
{deliberation_decision.get('next_goal')}")
            deliberation_decision['next_goal'] = None

        with self.agent.lock: # Lock for modifying agent.state.goals
            # 1. Add newly generated pending goals (if any) to agent's pending list
            newly_generated_pending_dicts = deliberation_decision.get('new_pending_goals', [])
            if isinstance(newly_generated_pending_dicts, list) and newly_generated_pending_dicts:
                current_pending_list = self.agent.state['goals'].setdefault('pending', [])
                for new_goal_dict in newly_generated_pending_dicts:
                    if isinstance(new_goal_dict, dict) and new_goal_dict.get('goal') and
new_goal_dict.get('priority'):
                        # Convert dict to Goal object, add defaults
                        new_goal_obj = Goal.from_dict(new_goal_dict)
                        new_goal_obj.status = GoalStatus.PENDING # Ensure status is pending
                        if not any(p.id == new_goal_obj.id for p in current_pending_list): # Avoid
duplicates based on ID
                            current_pending_list.append(new_goal_obj)
                            self.log.info(f"Added new goal '{new_goal_obj.goal[:50]}...' to pending list
from deliberation.")
                        else:
                            self.log.debug(f"Skipping duplicate new goal '{new_goal_obj.goal[:50]}...'
from deliberation.")
                    else:
                        self.log.warning(f"Deliberation proposed invalid new pending goal:
{new_goal_dict}")

            # 2. Handle selected 'next_goal'
            action_type = deliberation_decision.get('chosen_action_type')
            selected_next_goal_dict = deliberation_decision.get('next_goal') # This is a dict from
LLM

            # Current active goal (might be None)
            current_active_goal = self.agent.get_active_goal_object() # This is a Goal object or
None

            if action_type == 'pending_goal':
                pending_list_objs = [Goal.from_dict(g) for g in self.agent.state['goals'].get('pending',
[]) if isinstance(g, dict)] # Convert to objects for easier manipulation
                if selected_next_goal_dict and 'id' in selected_next_goal_dict:
                    found_idx = -1
```

```
                for i, pg_obj in enumerate(pending_list_objs):
                    if pg_obj.id == selected_next_goal_dict.get('id'):
                        found_idx = i
                        break
                if found_idx != -1:
                    selected_goal_obj = pending_list_objs.pop(found_idx) # Remove from pending
                    self.agent.state['goals']['active'] = selected_goal_obj.to_dict() # Set as active
                    selected_goal_obj.status = GoalStatus.ACTIVE
                    deliberation_decision['next_goal'] = selected_goal_obj # Update with full Goal
object
                    self.log.info(f"Moved pending goal {selected_goal_obj.id}
('{selected_goal_obj.goal[:50]}') to active.")
                else:
                    self.log.warning(f"LLM selected pending goal by ID
{selected_next_goal_dict.get('id')}, but not found in list. Idling.")
                    action_type = "idle" # Fallback to idle if selected pending goal not found
            elif pending_list_objs: # Fallback: LLM said pending but didn't specify, pop highest
                highest_priority_pending = pending_list_objs.pop(0) # Assumes sorted by priority
                self.agent.state['goals']['active'] = highest_priority_pending.to_dict()
                highest_priority_pending.status = GoalStatus.ACTIVE
                deliberation_decision['next_goal'] = highest_priority_pending
                self.log.info(f"Deliberation chose 'pending_goal' without specific ID; moved
highest priority '{highest_priority_pending.goal[:30]}...' to active.")
            else:
                self.log.warning("Deliberation chose 'pending_goal' but no pending goals
available. Idling.")
                action_type = "idle"

        elif action_type == 'new_goal' or action_type == 'idle_new_goal_generated':
            if selected_next_goal_dict and 'goal' in selected_next_goal_dict and 'priority' in
selected_next_goal_dict:
                new_active_goal_obj = Goal.from_dict(selected_next_goal_dict)
                new_active_goal_obj.status = GoalStatus.ACTIVE # Set directly as active
                self.agent.state['goals']['active'] = new_active_goal_obj.to_dict()
                deliberation_decision['next_goal'] = new_active_goal_obj # Update with full object
                self.log.info(f"Deliberation created and activated new goal:
{new_active_goal_obj.goal[:30]}...")
            else:
                self.log.warning("LLM chose 'new_goal' but 'next_goal' data was invalid. Idling.")
                action_type = "idle" # Fallback to idle

        elif action_type == 'resume_active_goal':
            if current_active_goal:
                deliberation_decision['next_goal'] = current_active_goal # Ensure it's the current
active
                current_active_goal.status = GoalStatus.ACTIVE # Re-affirm active status
                self.log.info(f"Deliberation chose to resume current active goal:
{current_active_goal.goal[:30]}...")
            else:
                self.log.warning("LLM chose 'resume_active_goal' but no active goal. Idling.")
                action_type = "idle"

        elif action_type in ['idle', 'reflection', 'self_assessment', 'external_command_action']:
```

```
            # If there was an active goal, it's being preempted. Archive it as 'PAUSED' or
'INTERRUPTED'.
            if current_active_goal:
                # For simplicity, mark as PAUSED. More sophisticated handling would be needed.
                current_active_goal.status = GoalStatus.PAUSED
                self.log.info(f"Current goal '{current_active_goal.goal[:30]}' PAUSED due to
{action_type}.")
                # Add back to pending. Ensure it's not a duplicate.
                pending_list_dicts = self.agent.state['goals'].setdefault('pending', [])
                if not any(g['id'] == current_active_goal.id for g in pending_list_dicts):
                    pending_list_dicts.insert(0, current_active_goal.to_dict()) # Put it back to
pending, maybe re-prioritize later
                else:
                    self.log.debug(f"Not re-adding paused goal {current_active_goal.id} to pending
as it's already there.")

            self.agent.state['goals']['active'] = None # Clear active goal if non-goal action
            deliberation_decision['next_goal'] = None # No goal object for these actions

        else: # Unknown action type from LLM
            self.log.warning(f"Unknown action type from deliberation: {action_type}. Defaulting to
Idle.")
            action_type = "idle" # Fallback to idle
            deliberation_decision['chosen_action_type'] = "idle"
            deliberation_decision['next_goal'] = None

        # Re-sort pending goals after any additions/removals, using the helper function
        self.agent.state['goals'].get('pending',[]).sort(key=get_priority_val, reverse=True)
        self.agent.save_state() # Save updated goal lists

    self.log.info(f"Deliberation complete. Chosen Action:
{deliberation_decision.get('chosen_action_type')}. Reason:
{deliberation_decision.get('reasoning','')[:100]}...")
    return deliberation_decision

def _act(self, current_goal_obj: Goal) -> bool:
    """
    Executes the current plan for the active_goal.
    Returns True if goal considered successfully processed for this cycle, False if critical error.
    """
    self.log.info(f"Acting on goal: {current_goal_obj.goal[:50]} (Plan steps:
{len(current_goal_obj.plan)})")
    plan_steps = current_goal_obj.plan

    if not plan_steps:
        self.log.warning("No steps in plan to execute.")
        # This might mean the goal is implicitly completed or failed if no plan.
        # For now, consider it a non-failure of the cycle itself.
        return True

    # Execute first step in the plan. The plan will be truncated or re-evaluated.
    # A more sophisticated execution manager would handle multi-step execution per cycle
    # or manage plan state (e.g., current step index).
    # For this model, assume one step (or one tool call) per `_act` call.
```

```
        step_to_execute = plan_steps[0] # Get the current step
        tool_name = step_to_execute.get("tool_name")
        params = step_to_execute.get("params", {})
        step_id = step_to_execute.get("step_id", "unknown_step")

        if not tool_name:
            self.log.error(f"Step {step_id} in plan for goal {current_goal_obj.id} has no tool_name.")
            # Mark this step as failed in the goal's context, then agent should replan or fail goal.
            current_goal_obj.outcome = "failed_step_execution"
            current_goal_obj.result_details = {"error": "Invalid plan step: no tool_name."}
            return False # Indicate failure for this cycle's processing of the goal

        try:
            # Tool execution
            step_exec_info = {"current_goal_id": current_goal_obj.id, "current_step_id": step_id,
"plan_step_details": step_to_execute}
            tool_result = self.agent.tool_manager.execute_tool(tool_name, params,
current_step_info=step_exec_info)

            # Record experience
            experience = Experience(
                triggering_goal_id=current_goal_obj.id,
                action_taken={"tool_name": tool_name, "params": params, "step_id": step_id},
                observation_result=tool_result,
                # Reward signal needs to be defined based on tool_result and goal progress
                reward_signal=self._calculate_reward(tool_result, current_goal_obj),
                internal_state_before=self.agent.self_model.beliefs, # Simplified snapshot
                internal_state_after=self.agent.self_model.beliefs # Needs update after action's
effects
            )
            self.agent.learning_module.add_experience(experience)

            # Process tool_result:
            # - Check for explicit goal completion from 'report_result' tool
            # - Check for sub-goal requests
            # - Check for errors that might require re-planning
            execution_info = tool_result.get('_exec_info', {})
            successful_execution = execution_info.get('execution_successful', False)

            if tool_name == "report_result": # This tool signals end of current goal's plan
                final_status = tool_result.get("status", "unknown")
                if final_status == "success":
                    current_goal_obj.status = GoalStatus.COMPLETED
                    current_goal_obj.outcome = "success"
                else: # failed, error, etc.
                    current_goal_obj.status = GoalStatus.FAILED
                    current_goal_obj.outcome = final_status
                current_goal_obj.result_details = tool_result
                current_goal_obj.completion_ts = datetime.now(timezone.utc).isoformat()
                current_goal_obj.plan = [] # Clear plan as it's finished.
                self.log.info(f"Goal '{current_goal_obj.goal[:50]}' processing finished by report_result.
Status: {final_status}")

            elif not successful_execution: # Tool execution itself failed
```

```
            self.log.warning(f"Tool '{tool_name}' execution failed. Error: {tool_result.get('error',
'Unknown error')}")
            # Trigger re-planning or goal failure
            new_plan_tuple = self.planning_module.replan_if_needed(current_goal_obj,
tool_result, observations[0] if observations else None) # Pass a relevant observation if available
            if new_plan_tuple:
                current_goal_obj.plan, current_goal_obj.thought = new_plan_tuple
                self.log.info(f"Successfully re-planned for goal {current_goal_obj.id}.")
            else: # Re-planning failed or not possible
                self.log.error(f"Failed to re-plan for goal {current_goal_obj.id} after tool failure. Goal
will likely fail.")
                current_goal_obj.status = GoalStatus.FAILED
                current_goal_obj.outcome = "failed_replan"
                current_goal_obj.plan = [] # Clear plan

        else: # Tool executed successfully, and it wasn't report_result
            # Remove the executed step from the plan
            if current_goal_obj.plan: # Ensure plan still exists
                current_goal_obj.plan.pop(0)
            if not current_goal_obj.plan: # If that was the last step, but not report_result
                self.log.warning(f"Plan for goal '{current_goal_obj.goal[:50]}' ended without
'report_result'. Goal might be incomplete.")
                # Agent might need to add a report_result step or re-evaluate.
                # For now, assume it needs to continue or will be handled by reflection.

        # Update the active goal in agent's state
        with self.agent.lock:
            if self.agent.state['goals'].get('active') and self.agent.state['goals']['active']['id'] ==
current_goal_obj.id:
                self.agent.state['goals']['active'] = current_goal_obj.to_dict()
                self.agent.save_state()
        return True

    except AgentError as e: # Catch errors from tool_manager.execute_tool or safety
violations
        self.log.error(f"AgentError during action execution for goal '{current_goal_obj.goal[:50]}':
{e}", exc_info=False)
        current_goal_obj.status = GoalStatus.FAILED
        current_goal_obj.outcome = f"AgentError: {type(e).__name__}"
        current_goal_obj.result_details = {"error": str(e), "error_type": type(e).__name__}
        current_goal_obj.plan = [] # Clear plan on critical error for this goal
        with self.agent.lock: # Update state
            if self.agent.state['goals'].get('active') and self.agent.state['goals']['active']['id'] ==
current_goal_obj.id:
                self.agent.state['goals']['active'] = current_goal_obj.to_dict()
                self.agent.save_state()
        self.agent.last_error = e # For main loop to see
        return False # Signal that the goal processing failed critically for this cycle


def _calculate_reward(self, tool_result: Dict, goal_obj: Goal) -> float:
    """Calculates a reward signal based on tool execution outcome and goal relevance."""
    # This is a placeholder. Reward shaping is a complex topic.
    reward = 0.0
```

```
        exec_info = tool_result.get('_exec_info', {})
        if exec_info.get('execution_successful'):
            reward += 0.1 # Small reward for successful tool use
            if tool_result.get('status') == 'success': # If tool indicates semantic success
                reward += 0.2
        else:
            reward -= 0.5 # Penalty for tool failure

        # Goal-related reward (conceptual)
        # if goal_obj.is_closer_to_completion(tool_result): reward += 0.5
        # if tool_result directly achieves a sub_goal of goal_obj: reward += 0.3
        if tool_result.get('status') == 'sub_goal_prepared': # Tool trying to advance goal via sub-
goaling
            reward += 0.1
        return round(reward, 2)


# --- Main AutonomousAgent Class (AGI Enhanced) ---
class AutonomousAgent:
    def __init__(self):
        self.log = get_logger("AGENT_CORE")
        self._status: str = "Booting" # Internal status, not SelfModel status
        self.lock = threading.Lock()
        self.state: Dict[str, Any] = {} # Agent's primary state dictionary
        self.goal_stack: List[Dict] = [] # Stack for managing sub-goals (parent goal data)
        self.cycle_count: int = 0
        self.last_error: Optional[Exception] = None
        self.current_goal_outcome: Optional[bool] = None # True for success/idle, False for failure
in cycle

        global _agent_instance_hack
        _agent_instance_hack = self # Set the global hack reference

        # Initialize core components
        self.llm_wrapper: BaseLLMWrapper # Type hint, initialized in _initialize_agent
        self.tool_manager: ToolExecutor
        self.memory_system: MemorySystem
        self.self_model: SelfModel
        self.cognitive_cycle: CognitiveCycle # Orchestrates perceive-act

        # AGI Modules
        self.perception_module: PerceptionModule # Already part of CognitiveCycle, but exposed
for direct access
        self.learning_module: LearningModule
        self.planning_module: PlanningModule # Already part of CognitiveCycle, but exposed for
direct access
        self.safety_module: SafetyModule
        self.embodiment: Optional[VirtualEmbodiment] = None # Or other embodiment interface
        self.comms_channel: Optional[FileChannel] = None # For multi-agent communication

        self.state['flags'] = {} # For inter-cycle communication or special states

        try:
            self._initialize_agent()
```

```
            self._update_status("Initialized")
            self.log.info(f"--- {AGENT_NAME} ({AGENT_VERSION}) Initialization Complete ---
Status: {self._status} ---")
            self.log.info(f"LLM Model: {LLM_MODEL_NAME_OR_PATH}, Device: {LLM_DEVICE}")
            self.log.info(f"Workspace: {WORKSPACE_DIR}")
            self.log.info(f"Max Context Tokens: {MAX_LLM_CONTEXT_TOKENS}, Max Response
Tokens: {MAX_LLM_RESPONSE_TOKENS}")
            self.log.warning(f"Shell Tool Enabled: {ENABLE_SHELL_TOOL}")
            self.log.warning(f"Code Generation Tool Enabled:
{ENABLE_CODE_GENERATION_TOOL}")
            self.log.warning(f"Self Modification Enabled: {ENABLE_SELF_MODIFICATION}")
            if ENABLE_SHELL_TOOL or ENABLE_CODE_GENERATION_TOOL or
ENABLE_SELF_MODIFICATION:
                self.log.critical("HIGH-RISK CAPABILITIES ARE ENABLED. RUN WITH EXTREME
CAUTION IN ISOLATED ENVIRONMENT.")
        except Exception as e_init:
            self.log.critical(f"CRITICAL UNHANDLED ERROR during agent initialization: {e_init}",
exc_info=True)
            self.shutdown() # Attempt partial shutdown
            raise ConfigurationError(f"Agent initialization failed critically: {e_init}") from e_init

    def _initialize_agent(self):
        self.log.info("Starting agent initialization sequence...")
        self._update_status("Initializing State")
        self.state = self._initialize_state() # Loads or creates default state
        self.cycle_count = self.state.get("cycle_count", 0)
        self.goal_stack = self.state.get("goal_stack", []) # Load goal stack

        # LLM Wrapper Initialization
        self._update_status("Initializing LLM")
        if LLM_MODEL_NAME_OR_PATH == "mock":
            self.llm_wrapper = MockLLMWrapper(LLM_MODEL_NAME_OR_PATH, LLM_DEVICE,
LLM_DEVICE_ID, MAX_LLM_CONTEXT_TOKENS, get_logger("LLM_WRAPPER"))
        elif LLM_MODEL_NAME_OR_PATH.startswith("gemini-"):
            if not GOOGLE_GENAI_AVAILABLE:
                raise ConfigurationError("Cannot use Gemini model: google-generativeai library not
installed.")
            self.llm_wrapper = GeminiLLMWrapper(LLM_MODEL_NAME_OR_PATH, LLM_DEVICE,
LLM_DEVICE_ID, MAX_LLM_CONTEXT_TOKENS, get_logger("LLM_WRAPPER"))
        elif TRANSFORMERS_AVAILABLE:
            global LLM_PIPELINE, LLM_TOKENIZER # Access global placeholders
            # if LLM_PIPELINE is None:
            #     # Ensure device is correctly mapped for transformers pipeline
            #     pipeline_device = LLM_DEVICE_ID if LLM_DEVICE in ['cuda', 'mps'] else -1 # -1 for
CPU
            #     LLM_TOKENIZER =
AutoTokenizer.from_pretrained(LLM_MODEL_NAME_OR_PATH, trust_remote_code=True)
            #     LLM_PIPELINE = pipeline(
            #         "text-generation",
            #         model=LLM_MODEL_NAME_OR_PATH,
            #         tokenizer=LLM_TOKENIZER,
            #         torch_dtype=torch.bfloat16 if TORCH_AVAILABLE else None, # Use bfloat16 if
torch available
            #         device=pipeline_device
```

```
            #     )
            self.llm_wrapper = TransformersLLMWrapper(LLM_MODEL_NAME_OR_PATH,
LLM_DEVICE, LLM_DEVICE_ID, MAX_LLM_CONTEXT_TOKENS,
get_logger("LLM_WRAPPER"))
        else: # Fallback to mock if specific not implemented here
            self.log.warning(f"LLM_MODEL '{LLM_MODEL_NAME_OR_PATH}' not fully configured
for wrapper selection, using Mock.")
            self.llm_wrapper = MockLLMWrapper(LLM_MODEL_NAME_OR_PATH, LLM_DEVICE,
LLM_DEVICE_ID, MAX_LLM_CONTEXT_TOKENS, get_logger("LLM_WRAPPER"))
        self.llm_wrapper._initialize_model() # Initialize the chosen LLM wrapper's model

        self._update_status("Initializing MemorySystem")
        self.memory_system = MemorySystem(self)

        self._update_status("Initializing SelfModel")
        self.self_model = SelfModel(self.state, DEFAULT_CORE_DIRECTIVES)

        self._update_status("Initializing ToolManager")
        self.tool_manager = ToolExecutor(self) # Tool discovery happens inside
        _init_self_mod_tools(self, self.tool_manager) # Initialize the SelfModificationTools handler
and register its UNSAFE methods

        # AGI Modules Initialization
        self._update_status("Initializing AGI Modules")
        self.learning_module = LearningModule(self)
        self.safety_module = SafetyModule(self)

        self._update_status("Initializing Embodiment")
        self.embodiment = VirtualEmbodiment(self) # Initialize the virtual embodiment
        self.log.info(f"Initialized VirtualEmbodiment. Current location: {self.embodiment.location}")

        self._update_status("Initializing Communication Channel")
        self.comms_channel = FileChannel(agent_id=AGENT_NAME,
shared_directory=str(AGENT_COMMS_DIR))
        self._setup_communication_handlers()

        self._update_status("Initializing CognitiveCycle")
        self.cognitive_cycle = CognitiveCycle(self)
        # Expose references from cognitive cycle for direct access (simplicity)
        self.perception_module = self.cognitive_cycle.perception_module
        self.planning_module = self.cognitive_cycle.planning_module

        # Other inits like playwright, resource monitor from base script if needed
        self._initialize_resource_monitor()
        self._initialize_playwright() # Only if tools require it, or always on.
        if self.tool_manager: self.tool_manager.check_playwright_browsers() # Check browsers for
browser tool

        self.log.info("Agent component initialization finished.")

    def _initialize_state(self) -> Dict[str, Any]:
        self.log.info(f"Initializing state from {STATE_FILE} or creating default.")
        if STATE_FILE.exists():
            try:
```

```
            with STATE_FILE.open('r') as f:
                state = json.load(f)
            # Basic validation and migration for new structure
            state.setdefault('goals', {'pending': [], 'active': None, 'completed': [], 'failed': []})
            for key in ['pending', 'completed', 'failed']: # Ensure these are lists
                if not isinstance(state['goals'].get(key), list): state['goals'][key] = []
            state.setdefault('cycle_count', 0)
            state.setdefault('knowledge_base', {}) # For SelfModel state and other persistent KB
items
            state.setdefault('goal_stack', []) # For sub-goal resumption
            state.setdefault('flags', {}) # For new flags system

            self.log.info("Agent state loaded successfully.")
            return state
        except json.JSONDecodeError as e:
            self.log.error(f"Error decoding state file {STATE_FILE}: {e}. Creating new state.")
        except Exception as e:
            self.log.error(f"Error loading state file {STATE_FILE}: {e}. Creating new state.")
    # Default state
    default_state = {
        "agent_name": AGENT_NAME,
        "agent_version": AGENT_VERSION,
        "cycle_count": 0,
        "goals": { # Standardized goal structure
            "pending": [], # List of Goal dicts
            "active": None, # Single Goal dict or None
            "completed": [], # List of Goal dicts
            "failed": [] # List of Goal dicts
        },
        "goal_stack": [], # List of parent Goal dicts when sub-goaling
        "knowledge_base": { # Contains self_model_state and other persistent KB
            "self_model_state": {} # SelfModel will populate this
        },
        "error_history_summary": [], # Summary of recent errors (SelfModel manages its view)
        "recent_successes_summary": [], # Summary of recent successes (SelfModel manages
its view)
        "recent_failures_summary": [], # Summary of recent failures (SelfModel manages its
view)
        "recent_tool_outcomes_summary": [], # Summary (SelfModel manages its view)
        "last_status": "Initialized",
        "flags": {} # For special agent states/triggers
    }
    self.log.info("Created new default agent state.")
    return default_state

  def save_state(self):
    self.log.debug(f"Saving agent state to {STATE_FILE}")
    with self.lock: # Ensure thread safety for state modification and saving
        # Prune lists in state if they grow too large (already handled for completed/failed in
_archive_goal)
        # Persist SelfModel's current state into the main state object
        if self.self_model:
            self.self_model.save_to_state(self.state) # SelfModel saves into
state['knowledge_base']['self_model_state']
```

```
        # Persist goal stack
        self.state['goal_stack'] = copy.deepcopy(self.goal_stack) # Save current goal stack
        # Persist summaries for SelfModel context on next load
        # These are views into potentially larger lists in memory system or main state lists
        self.state['error_history_summary'] = self.self_model.recent_errors
        self.state['recent_successes_summary'] = self.self_model.recent_successes
        self.state['recent_failures_summary'] = self.self_model.recent_failures
        self.state['recent_tool_outcomes_summary'] = self.self_model.recent_tool_outcomes
        self.state['last_status'] = self.self_model.current_status if self.self_model else
self._status

        try:
            temp_file = STATE_FILE.with_suffix(STATE_FILE.suffix + ".tmp")
            with temp_file.open('w') as f:
                json.dump(self.state, f, indent=2, default=str) # default=str for datetime etc.
            os.replace(temp_file, STATE_FILE) # Atomic replace
            self.log.info(f"Agent state saved successfully to {STATE_FILE}.")
        except Exception as e:
            self.log.error(f"Error saving agent state to {STATE_FILE}: {e}", exc_info=True)

    def _update_status(self, new_status: str):
        if self._status != new_status:
            self.log.info(f"Agent status changing from '{self._status}' to '{new_status}'")
            self._status = new_status
            if hasattr(self, 'self_model') and self.self_model: # Check if self_model initialized
                self.self_model.update_status(new_status) # Update SelfModel's view
            self.state["last_status"] = new_status # Persist to main state as well

    def get_active_goal_object(self) -> Optional[Goal]:
        """Returns the current active goal as a Goal object, or None."""
        active_goal_dict = self.state['goals'].get('active')
        if active_goal_dict and isinstance(active_goal_dict, dict):
            try:
                return Goal.from_dict(active_goal_dict)
            except Exception as e:
                self.log.error(f"Failed to convert active goal dict to Goal object: {e}. Dict:
{active_goal_dict}")
                return None
        return None

    def _create_metacognitive_goal(self, anomaly_description: str, priority: GoalPriority =
GoalPriority.CRITICAL, context: Optional[Dict]=None):
        self.log.warning(f"Creating metacognitive goal for: {anomaly_description}")
        meta_goal_id = f"goal_metacog_{uuid.uuid4()}"
        meta_goal_dict = Goal(
            id=meta_goal_id,
            goal=f"Address metacognitive anomaly: {anomaly_description}",
            status=GoalStatus.PENDING,
            priority=priority,
            origin="metacognitive_self_regulation",
            context=context or {"anomaly_details": anomaly_description},
            plan=[], # To be generated
            thought="Metacognitive goal created due to detected anomaly or system need."
```

```
        ).to_dict()
        with self.lock:
            pending_goals = self.state['goals'].setdefault('pending', [])
            pending_goals.insert(0, meta_goal_dict) # Insert at front for immediate consideration by
deliberation
            # Re-sort by priority just in case, though insert(0) + CRITICAL usually handles it
            def get_priority_val(g_dict_item): # Helper for sorting list of dicts
                p_val = g_dict_item.get('priority', GoalPriority.MEDIUM.value)
                if isinstance(p_val, GoalPriority): return p_val.value
                if isinstance(p_val, str): return GoalPriority[p_val.upper()].value
                return p_val # Assume it's already int value
            pending_goals.sort(key=get_priority_val, reverse=True)
            self.save_state() # Persist the new goal
        self.log.info(f"Created metacognitive goal {meta_goal_id} with {str(priority)} priority.")

    def _archive_goal(self, goal_data_dict: Dict, final_status_str: str):
        self.log.info(f"Archiving goal: {goal_data_dict.get('goal', 'N/A')[:50]} (ID:
{goal_data_dict.get('id')}) with status: {final_status_str}")
        goal_obj = Goal.from_dict(goal_data_dict) # Convert to object for easier handling
        goal_obj.status = GoalStatus(final_status_str) # Ensure enum type
        goal_obj.completion_ts = datetime.now(timezone.utc).isoformat()
        # goal_obj.outcome might have been set by report_result or failure handling
        # Store in completed/failed lists (in-memory state, potentially pruned)
        if goal_obj.status == GoalStatus.COMPLETED:
            self.state['goals'].setdefault('completed', []).append(goal_obj.to_dict())
            self.state['goals']['completed'] = self.state['goals']['completed'][-
MAX_COMPLETED_GOALS_IN_STATE:]
        else: # FAILED, CANCELLED
            self.state['goals'].setdefault('failed', []).append(goal_obj.to_dict())
            self.state['goals']['failed'] = self.state['goals']['failed'][-MAX_FAILED_GOALS_IN_STATE:]

        # Persist goal to long-term relational memory
        self.memory_system.add_memory_entry(goal_obj, persist_to_relational=True)

        # Record in SelfModel's recent successes/failures
        outcome_summary = {"goal_id": goal_obj.id, "goal_text": goal_obj.goal, "status":
str(goal_obj.status), "completion_ts": goal_obj.completion_ts, "replan_count":
goal_obj.replan_count}
        if goal_obj.status == GoalStatus.COMPLETED:
            self.self_model.recent_successes.append(outcome_summary)
            self.self_model.recent_successes = self.self_model.recent_successes[-10:]
        else:
            self.self_model.recent_failures.append(outcome_summary)
            self.self_model.recent_failures = self.self_model.recent_failures[-10:]

        # Handle Goal Stack Pop (Resume Parent Goal)
        with self.lock:
            active_goal_id = goal_data_dict.get('id')
            # Clear the current active goal slot IF it's the one we just archived
            current_active_in_state = self.state['goals'].get('active')
            if current_active_in_state and isinstance(current_active_in_state, dict) and
current_active_in_state.get('id') == active_goal_id:
                self.state['goals']['active'] = None
                self.log.debug(f"Archived goal {active_goal_id} was active, clearing active slot.")
```

```
            if self.goal_stack and goal_obj.parent_goal_id: # Only pop if this was a sub-goal from stack
                # Check if top of stack matches this goal's parent
                parent_snapshot = self.goal_stack[-1] # Peek
                if parent_snapshot.get('goal_data',{}).get('id') == goal_obj.parent_goal_id:
                    parent_goal_snapshot = self.goal_stack.pop()
                    parent_goal_data_dict = parent_goal_snapshot.get('goal_data')
                    if parent_goal_data_dict and isinstance(parent_goal_data_dict, dict):
                        self.state['goals']['active'] = parent_goal_data_dict # Resume parent
                        # Update parent's thought with sub-goal outcome
                        sub_goal_outcome_summary = f"Sub-goal '{goal_obj.goal[:30]}' (ID: {active_goal_id}) concluded with status: {final_status_str}."
                        parent_thought = parent_goal_data_dict.get('thought', "")
                        parent_goal_data_dict['thought'] = parent_thought + f"\n[Resuming after Sub-goal]: {sub_goal_outcome_summary}"
                        parent_goal_data_dict['status'] = GoalStatus.ACTIVE.value # Ensure parent is marked active
                        self.log.info(f"Popped parent goal '{parent_goal_data_dict.get('goal', 'Unknown Parent')[:50]}' (ID: {parent_goal_data_dict.get('id')}) from stack. Resuming.")
                        self._update_status(f"Resuming Parent Goal: {parent_goal_data_dict.get('goal', '')[:30]}")
                    else:
                        self.log.error("Popped invalid goal snapshot from stack.")
            elif not self.state['goals'].get('active'): # No active goal and stack is empty or current goal wasn't a sub-goal from stack
                self.log.info("Goal archived. No parent goal to resume from stack, or current goal was not a stacked sub-goal. Checking pending goals.")
                self._update_status("Idle (Post-Goal)")
            self.save_state() # Persist changes


    def run(self):
        self.log.info(f"--- {AGENT_NAME} ({AGENT_VERSION}) Run Loop Starting ---")
        if self._status != "Initialized": self._update_status("Idle") # Start in Idle state if not fresh init

        if not self.cognitive_cycle:
            self.log.critical("CognitiveCycle not initialized. Cannot run.")
            return

        if not self.state['goals'].get('active') and not self.state['goals'].get('pending'):
            self.log.info("No initial goals. Creating a default metacognitive goal to start.")
            self._create_metacognitive_goal("Initial system check, self-assessment, and environment exploration.", priority=GoalPriority.HIGH)

        self.last_agent_interaction_time = time.time() # For interactive mode

        while not STOP_SIGNAL_RECEIVED.is_set():
            self.cycle_count +=1
            self.state['cycle_count'] = self.cycle_count # Persist cycle_count

            active_goal_data_before_cycle = copy.deepcopy(self.state['goals'].get('active')) if self.state['goals'].get('active') else None
            self.last_error = None
            self.current_goal_outcome = None # Reset for this cycle
```

```
            cycle_ok = False
            try:
                # Main cognitive cycle execution
                cycle_ok = self.cognitive_cycle.run_cycle()
            except Exception as loop_err: # Should be caught within run_cycle, but as safeguard
                self.log.critical(f"CRITICAL UNHANDLED ERROR escaped cognitive cycle:
{loop_err}", exc_info=True)
                self.last_error = loop_err; self.current_goal_outcome = False
                STOP_SIGNAL_RECEIVED.set(); break # Stop on unhandled critical error

            if not cycle_ok and not STOP_SIGNAL_RECEIVED.is_set(): # Cycle itself reported failure
                self.log.critical("Cognitive cycle indicated critical failure. Stopping run loop.")
                STOP_SIGNAL_RECEIVED.set()
                break

            # Post-cycle processing
            if active_goal_data_before_cycle: # If there was an active goal
                # Determine final status of the goal processed in this cycle
                updated_active_goal_dict = self.state['goals'].get('active') # Check if it's still the same
goal

                goal_to_archive = None
                if updated_active_goal_dict and updated_active_goal_dict['id'] ==
active_goal_data_before_cycle['id']:
                    # If the goal is still active, its status inside the dict reflects outcome
                    goal_obj_after_cycle = Goal.from_dict(updated_active_goal_dict)
                    if goal_obj_after_cycle.status in [GoalStatus.COMPLETED, GoalStatus.FAILED,
GoalStatus.CANCELLED]:
                        goal_to_archive = updated_active_goal_dict
                elif not updated_active_goal_dict and active_goal_data_before_cycle.get('status') in
[GoalStatus.COMPLETED.value, GoalStatus.FAILED.value]:
                    # Goal was cleared from active and its status was terminal (e.g. by _archive_goal
during preemption)
                    goal_to_archive = active_goal_data_before_cycle
                elif self.current_goal_outcome is False and active_goal_data_before_cycle: # Cycle
failed while this goal was active
                    active_goal_data_before_cycle['status'] = GoalStatus.FAILED.value # Mark it as
failed
                    goal_to_archive = active_goal_data_before_cycle

                if goal_to_archive:
                    self._archive_goal(goal_to_archive, goal_to_archive['status'])
                # If current_goal_outcome is False but goal_to_archive is None, it implies cycle error
not specific to goal completion

            # Reflection (AGI Enhanced) - Can be more frequent or event-driven
            if self._should_reflect(active_goal_data_before_cycle):
                self._reflect_on_performance()

            # Save state periodically or after significant changes (already done in many places)
            # self.save_state() # Could do a final save here per cycle too.
            if self.state['flags'].get('re_evaluate_strategy_needed'):
                self.log.info("Flag set to re-evaluate strategy. Triggering metacognitive goal.")
```

```
            self._create_metacognitive_goal("Re-evaluate overall strategy due to significant
internal change (e.g., directive update).", priority=GoalPriority.HIGH)
            self.state['flags']['re_evaluate_strategy_needed'] = False

        # Sleep to prevent busy-looping if very idle
        if self._status == "Idle" and not self.state['goals'].get('active') and not
self.state['goals'].get('pending'):
            # Longer sleep if truly idle and no deliberation happened recently
            sleep_duration = IDLE_DELIBERATION_INTERVAL_SECONDS / 10 if time.time() -
LAST_DELIBERATION_TIME < IDLE_DELIBERATION_INTERVAL_SECONDS else 1.0
            time.sleep(max(0.1, sleep_duration))
        elif not STOP_SIGNAL_RECEIVED.is_set():
            time.sleep(0.05) # Short pause between cycles

    self.log.warning(f"--- {AGENT_NAME} Run Loop Exited ---")
    self.shutdown()

  def _should_reflect(self, processed_goal_data: Optional[Dict]) -> bool:
    # More nuanced reflection triggers
    if self.cycle_count % METACOGNITIVE_CHECK_INTERVAL_CYCLES == 0: # Reflect
every N cycles
        return True

    if processed_goal_data and Goal.from_dict(processed_goal_data).status in
[GoalStatus.COMPLETED, GoalStatus.FAILED]: # Reflect after significant goal outcome
        # Only reflect if enough goals processed since last time
        goals_processed_key = "goals_processed_since_reflection"
        self.state.setdefault(goals_processed_key, 0)
        self.state[goals_processed_key] +=1
        if self.state[goals_processed_key] >=
int(os.getenv("AGENT_REFLECTION_INTERVAL_GOALS", "3")):
            return True

    if time.time() - LAST_REFLECTION_TIME >
MANDATORY_REFLECTION_INTERVAL_SECONDS and self._status == "Idle":
        return True

    if self.state['flags'].get('explicit_reflection_requested'):
        return True

    return False

  @retry(attempts=2, delay=5)
  def _reflect_on_performance(self):
    global LAST_REFLECTION_TIME
    self.log.info("--- Reflecting on Performance ---")
    self.state['flags']['explicit_reflection_requested'] = False # Reset flag
    self.state["goals_processed_since_reflection"] = 0 # Reset counter
    try:
      # 1. Generate Self-Assessment Prompt
      assessment_prompt = self.self_model.get_self_assessment_prompt()
      # 2. Get LLM Assessment
      llm_assessment_str = self.llm_wrapper.generate(assessment_prompt,
max_new_tokens=2048, temperature=0.5) # Allow more tokens for detailed reflection
```

```
            reflection_data = extract_json_robust(llm_assessment_str)
            if reflection_data.get("error"):
                self.log.error(f"Failed to get valid JSON from LLM self-assessment:
{reflection_data.get('error')}")
                self.self_model.add_event_log(f"Self-assessment LLM call failed:
{reflection_data.get('error')}", event_type="error")
                return
            # 3. Update SelfModel and Knowledge Base
            updated_sm, updated_kb = self.self_model.update_from_reflection(reflection_data)
            # 4. Persist learned facts / prompt suggestions from reflection to MemorySystem
            if updated_kb:
                if isinstance(reflection_data.get('learned_facts'), list):
                    for fact_str in reflection_data['learned_facts']:
                        kf = KnowledgeFact(fact_statement=fact_str, metadata={"source":
"self_reflection"})
                        self.memory_system.add_memory_entry(kf, persist_to_vector=True,
persist_to_relational=True)
                    self.log.info(f"Added {len(reflection_data['learned_facts'])} learned facts to memory
from reflection.")

                if isinstance(reflection_data.get('prompt_tuning_suggestions'), list):
                    for sugg_str in reflection_data['prompt_tuning_suggestions']:
                        entry = BaseMemoryEntry(type="prompt_suggestion", content=sugg_str,
metadata={"source": "self_reflection"})
                        self.memory_system.add_memory_entry(entry, persist_to_vector=True)
                    self.log.info(f"Added {len(reflection_data['prompt_tuning_suggestions'])} prompt
suggestions to memory.")

            # 5. Act on critical findings from reflection (e.g. self_modification_needed)
            if reflection_data.get('self_modification_needed'):
                mod_desc = reflection_data['self_modification_needed']
                self.log.warning(f"Reflection identified need for self-modification: {mod_desc}")
                self._create_metacognitive_goal(
                    f"Investigate and potentially perform self-modification based on reflection:
{mod_desc}",
                    priority=GoalPriority.HIGH,
                    context={"modification_description": mod_desc, "source": "self_reflection"}
                )

            # 6. AGI: Audit directives based on reflection insights or periodically
            if self.cycle_count % int(os.getenv("DIRECTIVE_AUDIT_INTERVAL_CYCLES", "50")) ==
0: # Audit every N cycles
                audit_issues = self.safety_module.audit_directives_and_behavior()
                if audit_issues:
                    self.log.warning(f"Directive audit identified issues: {audit_issues}")
                    # Could create metacognitive goal to address audit findings
                    self._create_metacognitive_goal(f"Address directive audit findings:
{str(audit_issues)[:100]}", context={"audit_report": audit_issues})

            self.log.info("--- Reflection Complete ---")
        except LLMError as e:
            self.log.error(f"LLMError during reflection: {e}")
        except Exception as e:
            self.log.error(f"Unexpected error during reflection: {e}", exc_info=True)
```

```
        finally:
            LAST_REFLECTION_TIME = time.time()
            self._update_status("Idle")
            self.save_state()

    def _setup_communication_handlers(self):
        """Sets up handlers for different message types if comms_channel exists."""
        if self.comms_channel:
            self.comms_channel.register_handler(MessageType.QUERY,
self.handle_query_message)
            self.comms_channel.register_handler(MessageType.INFORM,
self.handle_inform_message)
            # Add more handlers for REQUEST_ACTION, HEARTBEAT etc.
            self.log.info("Basic communication handlers registered.")

    def handle_query_message(self, message: Message) -> Optional[Message]:
        self.log.info(f"Agent {self.agent_id} received QUERY from {message.sender_id}:
{message.content}")
        query_key = message.content.get("query_key")
        response_content = {}
        # Basic K/V store lookup for example
        if query_key and self.state.get('knowledge_base', {}).get(query_key):
            response_content = {"key": query_key, "value": self.state['knowledge_base']
[query_key], "status": "FOUND"}
        elif query_key:
            response_content = {"key": query_key, "value": None, "status": "NOT_FOUND"}
        else: # General query
            response_content = {"agent_status": self._status, "knowledge_summary_sample":
str(self.state.get('knowledge_base',{}))[:100]}
        return Message(sender_id=self.agent_id, receiver_id=message.sender_id,
type=MessageType.RESPONSE.value, content=response_content, correlation_id=message.id)

    def handle_inform_message(self, message: Message) -> None:
        self.log.info(f"Agent {self.agent_id} received INFORM from {message.sender_id}:
{message.content}")
        # Simple update to a general 'shared_knowledge' dict in state
        shared_knowledge = self.state.setdefault('shared_knowledge', {})
        inform_data = message.content.get("data", {})
        if isinstance(inform_data, dict):
            for k, v in inform_data.items():
                shared_knowledge[f"{message.sender_id}_{k}"] = v # Prefix with sender to avoid
clashes
        self.log.info(f"Updated knowledge from inform: {shared_knowledge}")
        self.save_state() # Save after learning
        # No direct response typically needed for INFORM

    def _initialize_resource_monitor(self):
        global RESOURCE_MONITOR
        if not PSUTIL_AVAILABLE:
            self.log.info("psutil not available, resource monitoring disabled.");
            return
        if RESOURCE_MONITOR: return
        self.log.info("Initializing resource monitor...")
        try:
```

```
            RESOURCE_MONITOR = psutil.Process(os.getpid())
            RESOURCE_MONITOR.cpu_percent(interval=None) # Initialize measurement
            self.log.info("Resource monitor initialized (psutil).")
        except Exception as e: self.log.error(f"Failed to initialize resource monitor: {e}");
RESOURCE_MONITOR = None

    def _initialize_playwright(self):
        global PLAYWRIGHT_INSTANCE, PLAYWRIGHT_BROWSER, PLAYWRIGHT_CONTEXT,
PLAYWRIGHT_PAGE
        if not PLAYWRIGHT_AVAILABLE:
            self.log.info("Playwright not available, skipping initialization.")
            return
        if PLAYWRIGHT_INSTANCE: return # Already initialized
        self.log.info("Initializing Playwright...")
        try:
            PLAYWRIGHT_INSTANCE = sync_playwright().start() # type: ignore
            PLAYWRIGHT_BROWSER =
PLAYWRIGHT_INSTANCE.chromium.launch(headless=True) # type: ignore
            PLAYWRIGHT_CONTEXT = PLAYWRIGHT_BROWSER.new_context( # type: ignore
                user_agent='Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36',
                java_script_enabled=True,
                ignore_https_errors=True
            )
            PLAYWRIGHT_PAGE = PLAYWRIGHT_CONTEXT.new_page() # type: ignore
            # Assign to agent instance variables as well (if they exist for some reason outside
globals)
            self.playwright_instance = PLAYWRIGHT_INSTANCE
            self.playwright_browser = PLAYWRIGHT_BROWSER
            self.playwright_context = PLAYWRIGHT_CONTEXT
            self.playwright_page = PLAYWRIGHT_PAGE
            self.log.info("Playwright initialized successfully (Chromium headless).")
        except Exception as e:
            self.log.error(f"Failed to initialize Playwright: {e}", exc_info=True)
            self._shutdown_playwright() # Attempt cleanup

    def _shutdown_playwright(self):
        global PLAYWRIGHT_INSTANCE, PLAYWRIGHT_BROWSER, PLAYWRIGHT_CONTEXT,
PLAYWRIGHT_PAGE
        if not PLAYWRIGHT_INSTANCE: return
        self.log.info("Shutting down Playwright...")
        with PLAYWRIGHT_LOCK:
            if PLAYWRIGHT_PAGE: try: PLAYWRIGHT_PAGE.close() # type: ignore # Added type
ignore
            except Exception: pass
            if PLAYWRIGHT_CONTEXT: try: PLAYWRIGHT_CONTEXT.close() # type: ignore
            except Exception: pass
            if PLAYWRIGHT_BROWSER: try: PLAYWRIGHT_BROWSER.close() # type: ignore
            except Exception: pass
            if PLAYWRIGHT_INSTANCE: try: PLAYWRIGHT_INSTANCE.stop() # type: ignore
            except Exception: pass
            PLAYWRIGHT_PAGE = None; PLAYWRIGHT_CONTEXT = None
            PLAYWRIGHT_BROWSER = None; PLAYWRIGHT_INSTANCE = None
            self.playwright_page = None; self.playwright_context = None
```

```
            self.playwright_browser = None; self.playwright_instance = None
            self.log.info("Playwright shutdown complete.")

    def _try_reset_playwright_page(self):
        if not PLAYWRIGHT_AVAILABLE or not self.playwright_context : return
        self.log.warning("Attempting to reset Playwright page...")
        with PLAYWRIGHT_LOCK:
            global PLAYWRIGHT_PAGE
            if self.playwright_page: try: self.playwright_page.close(); PLAYWRIGHT_PAGE = None;
# type: ignore
            except Exception: pass
            try:
                self.playwright_page = self.playwright_context.new_page() # type: ignore
                PLAYWRIGHT_PAGE = self.playwright_page # Update global
                self.log.info("Playwright page reset successfully.")
            except Exception as e:
                self.log.error(f"Failed to reset Playwright page: {e}", exc_info=True)
                self.playwright_page = None; PLAYWRIGHT_PAGE = None


# --- Signal Handling & Main Entry ---
def signal_handler(sig, frame):
    print(f"\nSignal {sig} received. Requesting graceful shutdown...")
    sig_log = get_logger("SIGNAL")
    sig_log.warning(f"Signal {sig} received. Setting stop signal.")
    STOP_SIGNAL_RECEIVED.set()
    time.sleep(1) # Give the agent a moment to shut down its loops
    # If _agent_instance_hack is set and shutdown hasn't fully completed, force it.
    if _agent_instance_hack and _agent_instance_hack._status != "Shutting Down":
        sig_log.warning("Forcing shutdown via signal handler as main loop might be stuck.")
        _agent_instance_hack.shutdown()
    else:
        # Exit more forcefully if needed after a timeout, but try to let logging flush
        if 'logging' in sys.modules: logging.shutdown()
        sys.exit(0) # Standard exit code for Ctrl+C

if __name__ == "__main__":
    nl = "\n" # For f-string clarity
    print(f"{nl}{'+'*70}{nl} Starting Agent: {AGENT_NAME} ({AGENT_VERSION}){nl} Workspace: "
          f"{WORKSPACE_DIR}{nl} LLM: {LLM_MODEL_NAME_OR_PATH} on {LLM_DEVICE}{nl}
Shell/"
          f"CodeGen/SelfMod Enabled: {ENABLE_SHELL_TOOL}/
{ENABLE_CODE_GENERATION_TOOL}/"
          f"{ENABLE_SELF_MODIFICATION}{nl} {'(USE WITH EXTREME CAUTION!)' if
ENABLE_SHELL_TOOL or ENABLE_CODE_GENERATION_TOOL or
ENABLE_SELF_MODIFICATION else ''}{nl}{'+'*70}")

    signal.signal(signal.SIGINT, signal_handler)
    signal.signal(signal.SIGTERM, signal_handler)

    main_agent_instance: Optional[AutonomousAgent] = None
    exit_code = 0
    try:
        # Handle CLI goal from COMMANDS_FILE
```

```
        initial_command_goal = None
        if COMMANDS_FILE.exists() and COMMANDS_FILE.stat().st_size > 0:
            try:
                command_text = COMMANDS_FILE.read_text().strip()
                if command_text:
                    # Log instance might not be available yet, so print to console
                    print(f"Found initial command in {COMMANDS_FILE}: {command_text[:100]}...")
                    initial_command_goal = Goal(goal=command_text, origin="cli_command_file",
priority=GoalPriority.HIGH).to_dict()
                    COMMANDS_FILE.write_text("") # Clear after reading
            except Exception as e_cmdfile:
                print(f"Error reading initial command file: {e_cmdfile}", file=sys.stderr)

        main_agent_instance = AutonomousAgent()

        # If an initial command goal was parsed, add it to pending goals
        if initial_command_goal and main_agent_instance:
            with main_agent_instance.lock:
                main_agent_instance.state['goals'].setdefault('pending',
[]).append(initial_command_goal)
                # Re-sort pending goals to ensure the new high-priority goal is considered
                def get_priority_val_for_sort(g_dict_item):
                    p_val = g_dict_item.get('priority', GoalPriority.MEDIUM.value)
                    if isinstance(p_val, GoalPriority): return p_val.value
                    if isinstance(p_val, str): return GoalPriority[p_val.upper()].value
                    return p_val
                main_agent_instance.state['goals']['pending'].sort(key=get_priority_val_for_sort,
reverse=True)
                main_agent_instance.save_state()
            main_agent_instance.log.info(f"Initial command goal '{initial_command_goal['goal']
[:50]}' added to pending goals.")

        # Start the agent's main run loop
        main_agent_instance.run()

    except ConfigurationError as cfg_err_main:
        print(f"\nFATAL CONFIGURATION ERROR: {cfg_err_main}", file=sys.stderr)
        if main_agent_instance and hasattr(main_agent_instance, 'log'):
main_agent_instance.log.critical(f"Agent failed to start due to ConfigurationError:
{cfg_err_main}", exc_info=True)
        else: logging.getLogger(AGENT_NAME).critical(f"Agent pre-init or init failed due to
ConfigurationError: {cfg_err_main}", exc_info=True)
        exit_code = 2
    except KeyboardInterrupt:
        print("\nMain process interrupted by user (KeyboardInterrupt).")
        if main_agent_instance and hasattr(main_agent_instance, 'log'):
main_agent_instance.log.warning("Main process caught KeyboardInterrupt.")
        else: logging.getLogger(AGENT_NAME).warning("Main process caught KeyboardInterrupt
during init/early phase.")
        exit_code = 130 # Standard exit code for Ctrl+C
    except Exception as main_exec_err:
        print(f"\nFATAL UNHANDLED ERROR in main execution: {main_exec_err}", file=sys.stderr)
        traceback.print_exc(file=sys.stderr)
```

```
        if main_agent_instance and hasattr(main_agent_instance, 'log'):
main_agent_instance.log.critical(f"Fatal unhandled error in main: {main_exec_err}",
exc_info=True)
        else: logging.getLogger(AGENT_NAME).critical(f"Fatal unhandled error during init/main:
{main_exec_err}", exc_info=True)
        exit_code = 1
    finally:
        if main_agent_instance and getattr(main_agent_instance, '_status', '') != "Shutting Down":
            print("\nEnsuring agent shutdown in main finally block...")
            if hasattr(main_agent_instance, 'log'): main_agent_instance.log.warning("Main finally
block ensuring agent shutdown.")
            main_agent_instance.shutdown()
        elif not main_agent_instance and 'main_log' in locals(): # If agent init failed badly
            main_log.warning("Agent instance likely not created or fully initialized. Basic
shutdown.") # type: ignore

        # Ensure global hack reference is cleared
        if _agent_instance_hack is not None:
            _agent_instance_hack = None # type: ignore

        print(f"--- {AGENT_NAME} Process Exiting (Code: {exit_code}) ---")
        if 'logging' in sys.modules: # Ensure logging is shut down
            logging.shutdown()
        sys.exit(exit_code)
```