```python
#!/usr/bin/env python3
"""
MONSTERDOG Robotics Orchestrator
===============================
A comprehensive orchestration script for robotics simulation, AI training,
and deployment workflows.

Features:
- Docker container management for Isaac Sim/ROS 2
- AGI node orchestration with vision-decision-action pipeline
- Metrics collection and monitoring integration
- CI/CD pipeline automation
- Reinforcement learning evaluation and training
- Multi-platform deployment (local, cloud, edge devices)

Usage:
    python monsterdog_orchestrator.py build          # Build Docker images
    python monsterdog_orchestrator.py simulate       # Run simulation
    python monsterdog_orchestrator.py train          # Run RL training
    python monsterdog_orchestrator.py deploy         # Deploy to target platform
    python monsterdog_orchestrator.py monitor        # Start monitoring services
"""

import argparse
import asyncio
import hashlib
import json
import logging
import os
import subprocess
import sys
import time
from datetime import datetime
from pathlib import Path
from typing import Dict, Any, Optional, List
import yaml

# Configuration
class Config:
    """Central configuration management"""

    def __init__(self, config_path: Optional[Path] = None):
        self.base_dir = Path(__file__).parent.resolve()
        self.config_path = config_path or self.base_dir / "config" / "monsterdog.yaml"
        self.log_dir = self.base_dir / "logs"
        self.models_dir = self.base_dir / "models"
        self.data_dir = self.base_dir / "data"

        # Create directories
        for directory in [self.log_dir, self.models_dir, self.data_dir]:
            directory.mkdir(parents=True, exist_ok=True)
        # Load configuration
        self.config = self._load_config()

        # Setup logging
        self._setup_logging()

    def _load_config(self) -> Dict[str, Any]:
        """Load configuration from YAML file"""
        default_config = {
            "docker": {
                "image_name": "monsterdog_robotics",
                "base_image": "nvidia/isaac-sim:2023.1.1",
                "gpu_enabled": True
            },
            "simulation": {
                "default_duration": 120,
                "ros_launch_file": "launch/isaac_sim_bridge.launch.py"
            },
            "agi": {
                "policy_path": "models/agi_policy.pth",
                "decision_frequency": 30.0  # Hz
            },
            "rl": {
                "environment": "gym_gazebo2:MARA-v1",
                "episodes": 50,
                "reward_threshold": 200.0
            },
            "monitoring": {
                "metrics_endpoint": "http://localhost:9090",
                "log_level": "INFO"
            },
            "deployment": {
                "platforms": ["local", "cybercortex", "agility_arc"],
                "default_platform": "local"
            }
        }

        if self.config_path.exists():
            try:
                with open(self.config_path, 'r') as f:
                    user_config = yaml.safe_load(f)
                # Merge configurations
                default_config.update(user_config)
            except Exception as e:
                logging.warning(f"Failed to load config from {self.config_path}: {e}")

        return default_config

    def _setup_logging(self):
        """Setup logging configuration"""
        log_level = getattr(logging, self.config["monitoring"]["log_level"].upper())
        logging.basicConfig(
            level=log_level,
            format='%(asctime)s | %(levelname)s | %(name)s | %(message)s',
            handlers=[
                logging.FileHandler(self.log_dir / "monsterdog.log"),
                logging.StreamHandler(sys.stdout)
            ]
        )

# Utilities
class Utils:
    """Utility functions for the orchestrator"""

    @staticmethod
    def calculate_file_hash(file_path: Path, algorithm: str = "sha256") -> str:
        """Calculate hash of a file"""
        hash_obj = hashlib.new(algorithm)
        with open(file_path, 'rb') as f:
            for chunk in iter(lambda: f.read(8192), b""):
                hash_obj.update(chunk)
        return hash_obj.hexdigest()

    @staticmethod
    def log_event(event: str, data: Dict[str, Any] = None, log_file: Path = None):
        """Log structured events to NDJSON format"""
        if log_file is None:
            log_file = Path("logs/events.ndjson")

        entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "event": event,
            "data": data or {}
        }

        with open(log_file, 'a') as f:
            f.write(json.dumps(entry) + '\n')

        logging.info(f"Event: {event} - {data}")

    @staticmethod
    def run_command(cmd: List[str], timeout: Optional[int] = None,
                   capture_output: bool = False) -> subprocess.CompletedProcess:
        """Run shell command with proper error handling"""
        try:
            result = subprocess.run(
                cmd,
                timeout=timeout,
                capture_output=capture_output,
                text=True,
                check=True
            )
            return result
        except subprocess.CalledProcessError as e:
            logging.error(f"Command failed: {' '.join(cmd)}")
            logging.error(f"Return code: {e.returncode}")
            if e.stdout:
                logging.error(f"STDOUT: {e.stdout}")
            if e.stderr:
                logging.error(f"STDERR: {e.stderr}")
            raise
        except subprocess.TimeoutExpired as e:
            logging.error(f"Command timed out: {' '.join(cmd)}")
            raise

# Core Components
class DockerManager:
    """Manages Docker operations for the robotics stack"""

    def __init__(self, config: Config):
        self.config = config
        self.docker_config = config.config["docker"]
        self.logger = logging.getLogger(__name__ + ".DockerManager")

    def build_image(self) -> None:
        """Build Docker image for robotics simulation"""
        self.logger.info("Building Docker image...")
        Utils.log_event("docker_build_start", {"image": self.docker_config["image_name"]})

        dockerfile_content = self._generate_dockerfile()
        dockerfile_path = self.config.base_dir / "Dockerfile"

        with open(dockerfile_path, 'w') as f:
            f.write(dockerfile_content)

        cmd = ["docker", "build", "-t", self.docker_config["image_name"], "."]
        Utils.run_command(cmd)

        Utils.log_event("docker_build_complete", {"image": self.docker_config["image_name"]})
        self.logger.info("Docker image built successfully")

    def run_simulation(self, duration: int = None) -> None:
        """Run simulation in Docker container"""
        duration = duration or self.config.config["simulation"]["default_duration"]

        self.logger.info(f"Starting simulation for {duration} seconds")
        Utils.log_event("simulation_start", {"duration": duration})

        cmd = [
            "docker", "run", "--rm",
            "-v", f"{self.config.log_dir}:/workspace/logs",
            "-v", f"{self.config.data_dir}:/workspace/data"
        ]

        if self.docker_config["gpu_enabled"]:
            cmd.extend(["--gpus", "all"])
        cmd.extend([
            self.docker_config["image_name"],
            "ros2", "launch", self.config.config["simulation"]["ros_launch_file"]
        ])

        try:
            Utils.run_command(cmd, timeout=duration)
        except subprocess.TimeoutExpired:
            self.logger.info("Simulation completed (timeout reached)")

        Utils.log_event("simulation_complete", {"duration": duration})

    def _generate_dockerfile(self) -> str:
        """Generate Dockerfile content"""
        return f'''FROM {self.docker_config["base_image"]}

# Install dependencies
RUN apt-get update && apt-get install -y \\
    python3-pip \\
    python3-dev \\
    ros-humble-desktop \\
    ros-humble-isaac-sim-interface \\
    && rm -rf /var/lib/apt/lists/*

# Install Python packages
COPY requirements.txt /tmp/
RUN pip3 install -r /tmp/requirements.txt

# Copy source code
COPY src/ /workspace/src/
COPY launch/ /workspace/launch/
COPY config/ /workspace/config/

WORKDIR /workspace

# Setup ROS environment
RUN echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
RUN echo "source /workspace/install/setup.bash" >> ~/.bashrc

# Build ROS packages
RUN . /opt/ros/humble/setup.sh && colcon build

CMD ["bash"]
'''

class AGINode:
    """AGI decision-making node for robotics control"""

    def __init__(self, config: Config):
        self.config = config
        self.agi_config = config.config["agi"]
        self.logger = logging.getLogger(__name__ + ".AGINode")
        self.policy_path = Path(self.agi_config["policy_path"])
    def initialize(self) -> None:
        """Initialize AGI node"""
        self.logger.info("Initializing AGI node...")
        Utils.log_event("agi_init_start")

        if not self.policy_path.exists():
            self.logger.warning(f"Policy file not found: {self.policy_path}")
            # Initialize with default policy
            self._create_default_policy()

        Utils.log_event("agi_init_complete", {"policy_path": str(self.policy_path)})

    def run_decision_cycle(self) -> Dict[str, Any]:
        """Run one decision cycle"""
        # Placeholder for actual AGI decision logic
        decision = {
            "timestamp": time.time(),
            "action": "forward",
            "confidence": 0.85,
            "observations": {
                "vision": "object_detected",
                "lidar": "clear_path",
                "imu": "stable"
            }
        }

        Utils.log_event("agi_decision", decision)
        return decision

    def _create_default_policy(self) -> None:
        """Create a default policy file"""
        self.policy_path.parent.mkdir(parents=True, exist_ok=True)
        # This would normally save a trained model
        with open(self.policy_path, 'w') as f:
            json.dump({"type": "default_policy", "version": "1.0"}, f)

class RLTrainer:
    """Reinforcement Learning training pipeline"""

    def __init__(self, config: Config):
        self.config = config
        self.rl_config = config.config["rl"]
        self.logger = logging.getLogger(__name__ + ".RLTrainer")

    def run_evaluation(self, episodes: int = None) -> Dict[str, float]:
        """Run RL evaluation"""
        episodes = episodes or self.rl_config["episodes"]

        self.logger.info(f"Running RL evaluation for {episodes} episodes")
        Utils.log_event("rl_eval_start", {"episodes": episodes})

        # Simulate RL evaluation (replace with actual RL code)
        import random
        rewards = [random.uniform(150, 300) for _ in range(episodes)]

        metrics = {
            "avg_reward": sum(rewards) / len(rewards),
            "max_reward": max(rewards),
            "min_reward": min(rewards),
            "episodes": episodes
        }

        Utils.log_event("rl_eval_complete", metrics)

        # Trigger retraining if performance is low
        if metrics["avg_reward"] < self.rl_config["reward_threshold"]:
            self.logger.warning("Low performance detected, triggering retraining")
            self._trigger_retraining()

        return metrics

    def _trigger_retraining(self) -> None:
        """Trigger model retraining"""
        Utils.log_event("rl_retrain_triggered")
        # Placeholder for retraining logic
        self.logger.info("Retraining initiated...")

class DeploymentManager:
    """Manages deployment to various platforms"""

    def __init__(self, config: Config):
        self.config = config
        self.deploy_config = config.config["deployment"]
        self.logger = logging.getLogger(__name__ + ".DeploymentManager")

    def deploy(self, platform: str = None) -> None:
        """Deploy to specified platform"""
        platform = platform or self.deploy_config["default_platform"]

        if platform not in self.deploy_config["platforms"]:
            raise ValueError(f"Unsupported platform: {platform}")

        self.logger.info(f"Deploying to platform: {platform}")
        Utils.log_event("deployment_start", {"platform": platform})

        if platform == "local":
            self._deploy_local()
        elif platform == "cybercortex":
            self._deploy_cybercortex()
        elif platform == "agility_arc":
            self._deploy_agility_arc()

        Utils.log_event("deployment_complete", {"platform": platform})

    def _deploy_local(self) -> None:
        """Deploy locally"""
        self.logger.info("Local deployment - copying files...")
        # Placeholder for local deployment logic

    def _deploy_cybercortex(self) -> None:
        """Deploy to CyberCortex platform"""
        self.logger.info("CyberCortex deployment...")
        # Placeholder for CyberCortex deployment logic

    def _deploy_agility_arc(self) -> None:
        """Deploy to Agility Arc platform"""
        self.logger.info("Agility Arc deployment...")
        # Placeholder for Agility Arc deployment logic

# Main Orchestrator
class MonsterdogOrchestrator:
    """Main orchestrator class"""

    def __init__(self, config_path: Optional[Path] = None):
        self.config = Config(config_path)
        self.docker_manager = DockerManager(self.config)
        self.agi_node = AGINode(self.config)
        self.rl_trainer = RLTrainer(self.config)
        self.deployment_manager = DeploymentManager(self.config)
        self.logger = logging.getLogger(__name__ + ".Orchestrator")

    def build(self) -> None:
        """Build the entire stack"""
        self.logger.info("Building MONSTERDOG stack...")
        self.docker_manager.build_image()
        self.agi_node.initialize()

    def simulate(self, duration: int = None) -> None:
        """Run simulation"""
        self.logger.info("Starting simulation...")
        self.docker_manager.run_simulation(duration)

    def train(self, episodes: int = None) -> Dict[str, float]:
        """Run training evaluation"""
        self.logger.info("Starting RL training evaluation...")
        return self.rl_trainer.run_evaluation(episodes)

    def deploy(self, platform: str = None) -> None:
        """Deploy to target platform"""
        self.logger.info("Starting deployment...")
        self.deployment_manager.deploy(platform)

    def monitor(self) -> None:
        """Start monitoring services"""
        self.logger.info("Starting monitoring services...")
        # Placeholder for monitoring logic
        Utils.log_event("monitoring_started")
    def run_full_pipeline(self) -> None:
        """Run the complete pipeline"""
        self.logger.info("Running full MONSTERDOG pipeline...")

        try:
            self.build()
            self.simulate()
            metrics = self.train()
            self.deploy()

            Utils.log_event("pipeline_complete", {"final_metrics": metrics})
            self.logger.info("Pipeline completed successfully")

        except Exception as e:
            Utils.log_event("pipeline_error", {"error": str(e)})
            self.logger.error(f"Pipeline failed: {e}")
            raise

def main():
    """Main CLI entry point"""
    parser = argparse.ArgumentParser(
        description="MONSTERDOG Robotics Orchestrator",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  %(prog)s build                     # Build Docker images and initialize
  %(prog)s simulate --duration 300 # Run 5-minute simulation
  %(prog)s train --episodes 100    # Run RL evaluation with 100 episodes
  %(prog)s deploy --platform local # Deploy to local environment
  %(prog)s pipeline                  # Run complete pipeline
        """
    )

    parser.add_argument("--config", type=Path, help="Configuration file path")
    parser.add_argument("--verbose", "-v", action="store_true", help="Verbose logging")

    subparsers = parser.add_subparsers(dest="command", required=True)

    # Build command
    subparsers.add_parser("build", help="Build Docker images and initialize components")

    # Simulate command
    sim_parser = subparsers.add_parser("simulate", help="Run robotics simulation")
    sim_parser.add_argument("--duration", type=int, help="Simulation duration in seconds")

    # Train command
    train_parser = subparsers.add_parser("train", help="Run RL training evaluation")
    train_parser.add_argument("--episodes", type=int, help="Number of episodes")

    # Deploy command
    deploy_parser = subparsers.add_parser("deploy", help="Deploy to target platform")
    deploy_parser.add_argument("--platform", choices=["local", "cybercortex", "agility_arc"],
                              help="Target deployment platform")
```

```python
    # Monitor command
    subparsers.add_parser("monitor", help="Start monitoring services")

    # Pipeline command
    subparsers.add_parser("pipeline", help="Run complete pipeline")

    args = parser.parse_args()

    # Initialize orchestrator
    orchestrator = MonsterdogOrchestrator(args.config)

    # Execute command
    try:
        if args.command == "build":
            orchestrator.build()
        elif args.command == "simulate":
            orchestrator.simulate(args.duration)
        elif args.command == "train":
            metrics = orchestrator.train(args.episodes)
            print(f"Training metrics: {json.dumps(metrics, indent=2)}")
        elif args.command == "deploy":
            orchestrator.deploy(args.platform)
        elif args.command == "monitor":
            orchestrator.monitor()
        elif args.command == "pipeline":
            orchestrator.run_full_pipeline()

    except Exception as e:
        logging.error(f"Command failed: {e}")
        sys.exit(1)

if __name__ == "__main__":
    main()
```