```
import json
from typing import List, Optional, Tuple

import numpy as np
from pydantic import BaseModel, Field
from tenacity import retry, stop_after_attempt, wait_exponential

from swarms.utils.auto_download_check_packages import (
    auto_check_and_download_package,
)
from swarms.utils.lazy_loader import lazy_import_decorator
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="swarm_matcher")


class SwarmType(BaseModel):
    name: str
    description: str
    embedding: Optional[List[float]] = Field(
        default=None, exclude=True
    )


class SwarmMatcherConfig(BaseModel):
    model_name: str = "sentence-transformers/all-MiniLM-L6-v2"
```

```
    embedding_dim: int = (
        512  # Dimension of the sentence-transformers model
    )


@lazy_import_decorator
class SwarmMatcher:
    """
    A class for matching tasks to swarm types based on their descriptions.
    It utilizes a transformer model to generate embeddings for task and swarm type descriptions,
    and then calculates the dot product to find the best match.
    """

    def __init__(self, config: SwarmMatcherConfig):
        """
        Initializes the SwarmMatcher with a configuration.

        Args:
            config (SwarmMatcherConfig): The configuration for the SwarmMatcher.
        """
        logger.add("swarm_matcher_debug.log", level="DEBUG")
        logger.debug("Initializing SwarmMatcher")

        try:
            import torch
        except ImportError:
```

```
    auto_check_and_download_package(
        "torch", package_manager="pip", upgrade=True
    )
    import torch


try:
    import transformers
except ImportError:
    auto_check_and_download_package(
        "transformers", package_manager="pip", upgrade=True
    )
    import transformers


self.torch = torch
try:
    self.config = config
    self.tokenizer = (
        transformers.AutoTokenizer.from_pretrained(
            config.model_name
        )
    )
    self.model = transformers.AutoModel.from_pretrained(
        config.model_name
    )
    self.swarm_types: List[SwarmType] = []
    logger.debug("SwarmMatcher initialized successfully")
```

```
    except Exception as e:
        logger.error(f"Error initializing SwarmMatcher: {str(e)}")
        raise

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10),
)
def get_embedding(self, text: str) -> np.ndarray:
    """
    Generates an embedding for a given text using the configured model.

    Args:
        text (str): The text for which to generate an embedding.

    Returns:
        np.ndarray: The embedding vector for the text.
    """
    logger.debug(f"Getting embedding for text: {text[:50]}...")
    try:
        inputs = self.tokenizer(
            text,
            return_tensors="pt",
            padding=True,
            truncation=True,
            max_length=512,
```

```
            )
            with self.torch.no_grad():
                outputs = self.model(**inputs)
            embedding = (
                outputs.last_hidden_state.mean(dim=1)
                .squeeze()
                .numpy()
            )
            logger.debug("Embedding generated successfully")
            return embedding
        except Exception as e:
            logger.error(f"Error generating embedding: {str(e)}")
            raise

    def add_swarm_type(self, swarm_type: SwarmType):
        """
        Adds a swarm type to the list of swarm types, generating an embedding for its description.

        Args:
            swarm_type (SwarmType): The swarm type to add.
        """
        logger.debug(f"Adding swarm type: {swarm_type.name}")
        try:
            embedding = self.get_embedding(swarm_type.description)
            swarm_type.embedding = embedding.tolist()
            self.swarm_types.append(swarm_type)
```

```
        logger.info(f"Added swarm type: {swarm_type.name}")
    except Exception as e:
        logger.error(
            f"Error adding swarm type {swarm_type.name}: {str(e)}"
        )
        raise

def find_best_match(self, task: str) -> Tuple[str, float]:
    """
    Finds the best match for a given task among the registered swarm types.

    Args:
        task (str): The task for which to find the best match.

    Returns:
        Tuple[str, float]: A tuple containing the name of the best matching swarm type and the score.
    """
    logger.debug(f"Finding best match for task: {task[:50]}...")
    try:
        task_embedding = self.get_embedding(task)
        best_match = None
        best_score = -float("inf")
        for swarm_type in self.swarm_types:
            score = np.dot(
                task_embedding, np.array(swarm_type.embedding)
            )
```

```
            if score > best_score:
                best_score = score
                best_match = swarm_type
        logger.info(
            f"Best match for task: {best_match.name} (score: {best_score})"
        )
        return best_match.name, float(best_score)
    except Exception as e:
        logger.error(
            f"Error finding best match for task: {str(e)}"
        )
        raise

def auto_select_swarm(self, task: str) -> str:
    """
    Automatically selects the best swarm type for a given task based on their descriptions.

    Args:
        task (str): The task for which to select a swarm type.

    Returns:
        str: The name of the selected swarm type.
    """
    logger.debug(f"Auto-selecting swarm for task: {task[:50]}...")
    best_match, score = self.find_best_match(task)
    logger.info(f"Task: {task}")
```

```
    logger.info(f"Selected Swarm Type: {best_match}")
    logger.info(f"Confidence Score: {score:.2f}")
    return best_match

def run_multiple(self, tasks: List[str], *args, **kwargs) -> str:
    swarms = []

    for task in tasks:
        output = self.auto_select_swarm(task)

        # Append
        swarms.append(output)

    return swarms

def save_swarm_types(self, filename: str):
    """
    Saves the registered swarm types to a JSON file.

    Args:
        filename (str): The name of the file to which to save the swarm types.
    """
    try:
        with open(filename, "w") as f:
            json.dump([st.dict() for st in self.swarm_types], f)
        logger.info(f"Saved swarm types to {filename}")
```

```
        except Exception as e:
            logger.error(f"Error saving swarm types: {str(e)}")
            raise

    def load_swarm_types(self, filename: str):
        """
        Loads swarm types from a JSON file.

        Args:
            filename (str): The name of the file from which to load the swarm types.
        """
        try:
            with open(filename, "r") as f:
                swarm_types_data = json.load(f)
            self.swarm_types = [
                SwarmType(**st) for st in swarm_types_data
            ]
            logger.info(f"Loaded swarm types from {filename}")
        except Exception as e:
            logger.error(f"Error loading swarm types: {str(e)}")
            raise


def initialize_swarm_types(matcher: SwarmMatcher):
    logger.debug("Initializing swarm types")
    swarm_types = [
```

SwarmType(

name="AgentRearrange",

description="Optimize agent order and rearrange flow for multi-step tasks, ensuring efficient task allocation and minimizing bottlenecks. Keywords: orchestration, coordination, pipeline optimization, task scheduling, resource allocation, workflow management, agent organization, process optimization",

),

SwarmType(

name="MixtureOfAgents",

description="Combine diverse expert agents for comprehensive analysis, fostering a collaborative approach to problem-solving and leveraging individual strengths. Keywords: multi-agent system, expert collaboration, distributed intelligence, collective problem solving, agent specialization, team coordination, hybrid approaches, knowledge synthesis",

),

SwarmType(

name="SpreadSheetSwarm",

description="Collaborative data processing and analysis in a spreadsheet-like environment, facilitating real-time data sharing and visualization. Keywords: data analysis, tabular processing, collaborative editing, data transformation, spreadsheet operations, data visualization, real-time collaboration, structured data",

),

SwarmType(

name="SequentialWorkflow",

description="Execute tasks in a step-by-step, sequential process workflow, ensuring a logical and methodical approach to task execution. Keywords: linear processing, waterfall methodology, step-by-step execution, ordered tasks, sequential operations, process flow, systematic approach,

staged execution",

),

SwarmType(

name="ConcurrentWorkflow",

description="Process multiple tasks or data sources concurrently in parallel, maximizing productivity and reducing processing time. Keywords: parallel processing, multi-threading, asynchronous execution, distributed computing, concurrent operations, simultaneous tasks, parallel workflows, scalable processing",

),

SwarmType(

name="HierarchicalSwarm",

description="Organize agents in a hierarchical structure with clear reporting lines and delegation of responsibilities. Keywords: management hierarchy, organizational structure, delegation, supervision, chain of command, tiered organization, structured coordination",

),

SwarmType(

name="AdaptiveSwarm",

description="Dynamically adjust agent behavior and swarm configuration based on task requirements and performance feedback. Keywords: dynamic adaptation, self-optimization, feedback loops, learning systems, flexible configuration, responsive behavior, adaptive algorithms",

),

SwarmType(

name="ConsensusSwarm",

description="Achieve group decisions through consensus mechanisms and voting protocols among multiple agents. Keywords: group decision making, voting systems, collective intelligence, agreement protocols, democratic processes, collaborative decisions",

```
        # ),
    ]

    for swarm_type in swarm_types:
        matcher.add_swarm_type(swarm_type)
    logger.debug("Swarm types initialized")



@lazy_import_decorator
def swarm_matcher(task: str, *args, **kwargs):
    """
    Runs the SwarmMatcher example with predefined tasks and swarm types.
    """
    config = SwarmMatcherConfig()
    matcher = SwarmMatcher(config)
    initialize_swarm_types(matcher)

    # matcher.save_swarm_types(f"swarm_logs/{uuid4().hex}.json")

    swarm_type = matcher.auto_select_swarm(task)

    logger.info(f"{swarm_type}")

    return swarm_type
```

```
# from typing import List, Tuple, Dict
# from pydantic import BaseModel, Field
# from loguru import logger
# from uuid import uuid4
# import chromadb
# import json
# from tenacity import retry, stop_after_attempt, wait_exponential


# class SwarmType(BaseModel):
#     """A swarm type with its name, description and optional metadata"""

#     id: str = Field(default_factory=lambda: str(uuid4()))
#     name: str
#     description: str
#     metadata: Dict = Field(default_factory=dict)


# class SwarmMatcherConfig(BaseModel):
#     """Configuration for the SwarmMatcher"""

#     collection_name: str = "swarm_types"
#     distance_metric: str = "cosine"  # or "l2" or "ip"
#     embedding_function: str = (
#         "sentence-transformers/all-mpnet-base-v2"  # Better model than MiniLM
#     )
```

```
#     persist_directory: str = "./chroma_db"



# class SwarmMatcher:
#     """
#     An improved swarm matcher that uses ChromaDB for better vector similarity search.
#     Features:
#     - Persistent storage of embeddings
#     - Better vector similarity search with multiple distance metrics
#     - Improved embedding model
#     - Metadata filtering capabilities
#     - Batch operations support
#     """

#     def __init__(self, config: SwarmMatcherConfig):
#         """Initialize the improved swarm matcher"""
#         logger.add("swarm_matcher.log", rotation="100 MB")
#         self.config = config

#         # Initialize ChromaDB client with persistence
#         self.chroma_client = chromadb.Client()

#         # Get or create collection
#         try:
#             self.collection = self.chroma_client.get_collection(
#                 name=config.collection_name,
```

```
#             )
#         except ValueError:
#             self.collection = self.chroma_client.create_collection(
#                 name=config.collection_name,
#                 metadata={"hnsw:space": config.distance_metric},
#             )

#         logger.info(
#             f"Initialized SwarmMatcher with collection '{config.collection_name}'"
#         )

#     def add_swarm_type(self, swarm_type: SwarmType) -> None:
#         """Add a single swarm type to the collection"""
#         try:
#             self.collection.add(
#                 ids=[swarm_type.id],
#                 documents=[swarm_type.description],
#                 metadatas=[
#                     {"name": swarm_type.name, **swarm_type.metadata}
#                 ],
#             )
#             logger.info(f"Added swarm type: {swarm_type.name}")
#         except Exception as e:
#             logger.error(
#                 f"Error adding swarm type {swarm_type.name}: {str(e)}"
#             )
```

```
#             raise

#     def add_swarm_types(self, swarm_types: List[SwarmType]) -> None:
#         """Add multiple swarm types in batch"""
#         try:
#             self.collection.add(
#                 ids=[st.id for st in swarm_types],
#                 documents=[st.description for st in swarm_types],
#                 metadatas=[
#                     {"name": st.name, **st.metadata}
#                     for st in swarm_types
#                 ],
#             )
#             logger.info(f"Added {len(swarm_types)} swarm types")
#         except Exception as e:
#             logger.error(
#                 f"Error adding swarm types in batch: {str(e)}"
#             )
#             raise

#     @retry(
#         stop=stop_after_attempt(3),
#         wait=wait_exponential(multiplier=1, min=4, max=10),
#     )
#     def find_best_matches(
#         self,
```

```
#         task: str,
#         n_results: int = 3,
#         score_threshold: float = 0.7,
#     ) -> List[Tuple[str, float]]:
#         """
#         Find the best matching swarm types for a given task
#         Returns multiple matches with their scores
#         """
#         try:
#             results = self.collection.query(
#                 query_texts=[task],
#                 n_results=n_results,
#                 include=["metadatas", "distances"],
#             )

#             matches = []
#             for metadata, distance in zip(
#                 results["metadatas"][0], results["distances"][0]
#             ):
#                 # Convert distance to similarity score (1 - normalized_distance)
#                 score = 1 - (
#                     distance / 2
#                 )  # Normalize cosine distance to [0,1]
#                 if score >= score_threshold:
#                     matches.append((metadata["name"], score))
```

```
#             logger.info(f"Found {len(matches)} matches for task")
#             return matches

#         except Exception as e:
#             logger.error(f"Error finding matches for task: {str(e)}")
#             raise

#     def auto_select_swarm(self, task: str) -> str:
#         """
#         Automatically select the best swarm type for a task
#         Returns only the top match
#         """
#         matches = self.find_best_matches(task, n_results=1)
#         if not matches:
#             logger.warning("No suitable matches found for task")
#             return "SequentialWorkflow"  # Default fallback

#         best_match, score = matches[0]
#         logger.info(
#             f"Selected swarm type '{best_match}' with confidence {score:.3f}"
#         )
#         return best_match

#     def run_multiple(self, tasks: List[str]) -> List[str]:
#         """Process multiple tasks in batch"""
#         return [self.auto_select_swarm(task) for task in tasks]
```

```
#     def save_swarm_types(self, filename: str) -> None:
#         """Export swarm types to JSON"""
#         try:
#             all_data = self.collection.get(
#                 include=["metadatas", "documents"]
#             )
#             swarm_types = [
#                 SwarmType(
#                     id=id_,
#                     name=metadata["name"],
#                     description=document,
#                     metadata={
#                         k: v
#                         for k, v in metadata.items()
#                         if k != "name"
#                     },
#                 )
#                 for id_, metadata, document in zip(
#                     all_data["ids"],
#                     all_data["metadatas"],
#                     all_data["documents"],
#                 )
#             ]

#             with open(filename, "w") as f:
```

```
#                 json.dump(
#                     [st.dict() for st in swarm_types], f, indent=2
#                 )
#             logger.info(f"Saved swarm types to {filename}")
#         except Exception as e:
#             logger.error(f"Error saving swarm types: {str(e)}")
#             raise

#     def load_swarm_types(self, filename: str) -> None:
#         """Import swarm types from JSON"""
#         try:
#             with open(filename, "r") as f:
#                 swarm_types_data = json.load(f)
#             swarm_types = [SwarmType(**st) for st in swarm_types_data]
#             self.add_swarm_types(swarm_types)
#             logger.info(f"Loaded swarm types from {filename}")
#         except Exception as e:
#             logger.error(f"Error loading swarm types: {str(e)}")
#             raise


# def initialize_default_swarm_types(matcher: SwarmMatcher) -> None:
#     """Initialize the matcher with default swarm types"""
#     swarm_types = [
#         SwarmType(
#             name="AgentRearrange",
```

```
#             description="""
#                 Optimize agent order and rearrange flow for multi-step tasks, ensuring efficient task
allocation
#                     and minimizing bottlenecks. Specialized in orchestration, coordination, pipeline
optimization,
#                 task scheduling, resource allocation, workflow management, agent organization, and
process optimization.
#             Best for tasks requiring complex agent interactions and workflow optimization.
#             """,
#             metadata={
#                 "category": "optimization",
#                 "complexity": "high",
#             },
#         ),
#         SwarmType(
#             name="MixtureOfAgents",
#             description="""
#                 Combine diverse expert agents for comprehensive analysis, fostering a collaborative
approach
#             to problem-solving and leveraging individual strengths. Focuses on multi-agent systems,
#                     expert collaboration, distributed intelligence, collective problem solving, agent
specialization,
#                 team coordination, hybrid approaches, and knowledge synthesis. Ideal for complex
problems
#             requiring multiple areas of expertise.
#             """,
```

```
#             metadata={
#                 "category": "collaboration",
#                 "complexity": "high",
#             },
#         ),
#         SwarmType(
#             name="SpreadSheetSwarm",
#             description="""
#             Collaborative data processing and analysis in a spreadsheet-like environment, facilitating
#             real-time data sharing and visualization. Specializes in data analysis, tabular processing,
#             collaborative editing, data transformation, spreadsheet operations, data visualization,
#             real-time collaboration, and structured data handling. Perfect for data-intensive tasks
#             requiring structured analysis.
#             """,
#             metadata={
#                 "category": "data_processing",
#                 "complexity": "medium",
#             },
#         ),
#         SwarmType(
#             name="SequentialWorkflow",
#             description="""
#                 Execute tasks in a step-by-step, sequential process workflow, ensuring a logical and methodical
#                 approach to task execution. Focuses on linear processing, waterfall methodology, step-by-step
```

```
#             execution, ordered tasks, sequential operations, process flow, systematic approach, and
staged
#             execution. Best for tasks requiring strict order and dependencies.
#             """,
#             metadata={"category": "workflow", "complexity": "low"},
#         ),
#         SwarmType(
#             name="ConcurrentWorkflow",
#             description="""
#             Process multiple tasks or data sources concurrently in parallel, maximizing productivity
#             and reducing processing time. Specializes in parallel processing, multi-threading,
#                 asynchronous execution, distributed computing, concurrent operations, simultaneous
tasks,
#             parallel workflows, and scalable processing. Ideal for independent tasks that can be
#             processed simultaneously.
#             """,
#             metadata={"category": "workflow", "complexity": "medium"},
#         ),
#     ]

#     matcher.add_swarm_types(swarm_types)
#     logger.info("Initialized default swarm types")


# def create_swarm_matcher(
#     persist_dir: str = "./chroma_db",
```

```
#     collection_name: str = "swarm_types",
# ) -> SwarmMatcher:
#     """Convenience function to create and initialize a swarm matcher"""
#     config = SwarmMatcherConfig(
#         persist_directory=persist_dir, collection_name=collection_name
#     )
#     matcher = SwarmMatcher(config)
#     initialize_default_swarm_types(matcher)
#     return matcher


# # Example usage
# def swarm_matcher(task: str) -> str:
#     # Create and initialize matcher
#     matcher = create_swarm_matcher()

#     swarm_type = matcher.auto_select_swarm(task)
#     print(f"Task: {task}\nSelected Swarm: {swarm_type}\n")

#     return swarm_type


# # # Example usage
# # if __name__ == "__main__":
# #     # Create and initialize matcher
# #     matcher = create_swarm_matcher()
```

```
# #     # Example tasks
# #     tasks = [
# #         "Analyze this spreadsheet of sales data and create visualizations",
# #         "Coordinate multiple AI agents to solve a complex problem",
# #         "Process these tasks one after another in a specific order",
# #         "Write multiple blog posts about the latest advancements in swarm intelligence all at once",
# #         "Write a blog post about the latest advancements in swarm intelligence",
# #     ]

# #     # Process tasks
# #     for task in tasks:
# #         swarm_type = matcher.auto_select_swarm(task)
# #         print(f"Task: {task}\nSelected Swarm: {swarm_type}\n")
```