ДОДАТОК А

Вихідний код для методів на основі інженерії контексту

Лістинг А.1 Реалізація базового сценарію

```
class TaskEvent(Event):
    state: TaskState

class GenerationEvent(TaskEvent):
    pass

class EvaluationEvent(TaskEvent):
    pass

class FinalizationEvent(TaskEvent):
    pass

class SimpleAgentWorkflow(Workflow):
    def __init__(self, model, attempts, system_prompt,
verbose=False, timeout=1500, **kwargs):
        super().__init__(timeout=timeout, **kwargs)
        self.model = model
        self.attempts = attempts
        self.system_prompt = system_prompt
        self.verbose = verbose

    async def snapshot(self, ctx: Context):
        agent_state = await ctx.store.get("agent_state")
        return agent_state.copy()

    async def update_prompts_pile(self, ctx: Context,
conversation):
        all_prompts = await ctx.store.get("all_prompts", [])
        all_prompts.append(copy.deepcopy(conversation))
        await ctx.store.set("all_prompts", all_prompts)
```

```
    async def update_completions_pile(self, ctx: Context,
conversation):
        all_completions = await
ctx.store.get("all_completions", [])
        all_completions.append(copy.deepcopy(conversation))
        await ctx.store.set("all_completions", all_completions)

    @step
    async def initial_step(self, ctx: Context, ev: StartEvent)
-> GenerationEvent:
        state = ev.state
        message = ev.message

        state.conversation = [self.system_prompt, message]
        state.agent_states = []

        state.conversation.append(message)
        await ctx.store.set("agent_state", state.conversation)

        snapshot = await self.snapshot(ctx)
        state.agent_states.append(snapshot)

        return GenerationEvent(state=state)

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=10,
max_attempts=10))
    async def generation_step(self, ctx: Context, ev:
GenerationEvent) -> EvaluationEvent:
        state = ev.state
        conversation = state.conversation

        snapshot = await self.snapshot(ctx)
        state.agent_states.append(snapshot)

        def role_mapping(role):
            if role == 'system':
```

```
                return MessageRole.SYSTEM
            elif role == 'user':
                return MessageRole.USER
            elif role == 'assistant':
                return MessageRole.ASSISTANT

        llamaindex_conversation = [
            ChatMessage(role=role_mapping(m['role']),
content=m['content']) for m in conversation
        ]

        await self.update_prompts_pile(ctx, conversation)
        response = await model.achat(llamaindex_conversation)

        response_message = {'role': 'assistant', 'content':
response.message.content}
        state.conversation.append(response_message)
        await                self.update_completions_pile(ctx,
[response_message])
        await ctx.store.set("agent_state", state.conversation)

        return EvaluationEvent(state=state)

    @step
    async def feedback_step(self, ctx: Context, ev:
EvaluationEvent) -> GenerationEvent | FinalizationEvent:
        state = ev.state

        result        =        evaluate_task(state.task_data,
state.conversation, verbose=self.verbose)
        state.attempts += 1

        is_correct = result['is_correct']

        if is_correct or state.attempts >= self.attempts:
            state.is_done = True
```

```
            state.stats = {
              'pass@1': result.get('pass@1', 0),
              'score': result.get('score', 0),
              'best_score':   max(state.stats.get('best_score',
0), result.get('score', 0)),
              'correct': is_correct
            }
            return FinalizationEvent(state=state)
        else:
            message    =    {"role":    "user",    "content":
str(result['response'])}
            state.conversation.append(message)

            return GenerationEvent(state=state)

    @step
    async    def    finish_step(self,    ctx:    Context,    ev:
FinalizationEvent) -> StopEvent:
        state = ev.state

        all_prompts = await ctx.store.get("all_prompts", [])
        state.total_prompt_tokens = recount_tokens(all_prompts)

        all_completions                =                await
ctx.store.get("all_completions", [])
        state.total_completion_tokens                          =
recount_tokens(all_completions)

        return StopEvent(result=state)
```

Лістинг А.2 Реалізація сценарію за методом Reflexion

```
class ReflexionTaskEvent(Event):
    state: TaskState

class ReflexionGenerationEvent(ReflexionTaskEvent):
```

```
    pass

class ReflexionEvaluationEvent(ReflexionTaskEvent):
    pass

class ReflexionRefinementEvent(ReflexionTaskEvent):
    pass

class ReflexionFinalizationEvent(ReflexionTaskEvent):
    pass

class ReflexionAgentWorkflow(Workflow):
    def __init__(self, model, attempts, system_prompt,
verbose=False, timeout=1200, **kwargs):
        super().__init__(timeout=timeout, **kwargs)
        self.model = model
        self.attempts = attempts
        self.system_prompt = system_prompt
        self.verbose = verbose

    async def snapshot(self, ctx: Context):
        agent_state = await ctx.store.get("agent_state")
        return agent_state.copy()

    async def update_prompts_pile(self, ctx: Context,
conversation):
        all_prompts = await ctx.store.get("all_prompts", [])
        all_prompts.append(copy.deepcopy(conversation))
        await ctx.store.set("all_prompts", all_prompts)

    async def update_completions_pile(self, ctx: Context,
conversation):
        all_completions = await
ctx.store.get("all_completions", [])
        all_completions.append(copy.deepcopy(conversation))
        await ctx.store.set("all_completions", all_completions)
```

```
    @step
    async def initial_step(self, ctx: Context, ev: StartEvent)
-> ReflexionGenerationEvent:
        state = ev.state
        message = ev.message

        state.conversation = [self.system_prompt, message]
        state.agent_states = []

        await                       ctx.store.set("agent_state",
state.conversation.copy())

        return ReflexionGenerationEvent(state=state)

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
    async   def   generation_step(self,   ctx:   Context,   ev:
ReflexionGenerationEvent) -> ReflexionEvaluationEvent:
        state = ev.state
        conversation = state.conversation
        agent_state = await ctx.store.get("agent_state")

        snapshot = await self.snapshot(ctx)
        state.agent_states.append(snapshot)

        def role_mapping(role):
            if role == 'system':
                return MessageRole.SYSTEM
            elif role == 'user':
                return MessageRole.USER
            elif role == 'assistant':
                return MessageRole.ASSISTANT

        llamaindex_conversation = [
```

```
            ChatMessage(role=role_mapping(m['role']),
content=m['content']) for m in agent_state
        ]

        await self.update_prompts_pile(ctx, agent_state)
        response = await model.achat(llamaindex_conversation)

        response_message = {'role': 'assistant', 'content':
response.message.content}
        state.conversation.append(response_message)
        agent_state.append(response_message)
        await                self.update_completions_pile(ctx,
[response_message])

        await ctx.store.set("agent_state", agent_state.copy())

        return ReflexionEvaluationEvent(state=state)

    @step
    async def feedback_step(self, ctx: Context, ev:
ReflexionEvaluationEvent) -> ReflexionRefinementEvent |
ReflexionFinalizationEvent:
        state = ev.state

        result        =        evaluate_task(state.task_data,
state.conversation, verbose=self.verbose)
        state.attempts += 1

        is_correct = result['is_correct']

        if is_correct or state.attempts >= self.attempts:
            state.is_done = True
            state.stats = {
              'pass@1': result.get('pass@1', 0),
              'score': result.get('score', 0),
```

```
            'best_score':   max(state.stats.get('best_score',
0), result.get('score', 0)),
            'correct': is_correct
        }
        return ReflexionFinalizationEvent(state=state)
    else:
        message    =    {"role":    "user",    "content":
str(result['response'])}
        state.conversation.append(message)

        return ReflexionRefinementEvent(state=state)

@step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
async    def    refine_step(self,    ctx:    Context,    ev:
ReflexionRefinementEvent) -> ReflexionGenerationEvent:
    state = ev.state

    request = state.conversation[1]['content']
    answer = state.conversation[-2]['content']
    system_feedback = state.conversation[-1]['content']

    validator_prompt = f"""
You need to provide a natural language feedback to proposed
answer to a user request.
For tasks like coding or math you may also get a response of a
testing suite or a proofer as a SYSTEM FEEDBACK that will mention
the ERRORS.
Provide a helpful and understandable feedback and follow these
rules:
* BE CONCISE
* Follow the SYSTEM FEEDBACK over your judgement
* Try to decompose answer into relevant aspects and evaluate
each of them and then total
* You may use the scoring system to highlight good, fine and
critical to fix parts of the solution
```

```
* You are encouraged to propose the fix ideas, BUT YOU SHOULD NOT PROVIDE YOUR OWN SOLUTIONS, REVISIONS OR SIMILAR

REQUEST:
{request}

ANSWER:
{answer}

SYSTEM FEEDBACK:
{system_feedback}
        """

        llamaindex_conversation = [
            {'role': 'user', 'content': validator_prompt},
            {'role': 'assistant', 'content': 'FEEDBACK:'}
        ]

        llamaindex_conversation = [
            ChatMessage(**m) for m in llamaindex_conversation
        ]

        response = await model.achat(llamaindex_conversation)

        message = {"role": "user", "content": response.message.content + "\n" + "Please, generate a refined version of the solution."}
        state.conversation.append(message)
        await self.update_prompts_pile(ctx, [{"role": "user", "content": validator_prompt}])
        await self.update_completions_pile(ctx, [message])

        agent_state = await ctx.store.get("agent_state")
        agent_state.append(message)
        await ctx.store.set("agent_state", agent_state.copy())
```

```
        return ReflexionGenerationEvent(state=state)

    @step
    async def finish_step(self, ctx: Context, ev:
ReflexionFinalizationEvent) -> StopEvent:
        state = ev.state

        all_prompts = await ctx.store.get("all_prompts", [])
        state.total_prompt_tokens = recount_tokens(all_prompts)

        all_completions = await
ctx.store.get("all_completions", [])
        state.total_completion_tokens =
recount_tokens(all_completions)

        return StopEvent(result=state)
```

Лістинг А.3 Реалізація сценарію за методом LATS

```
class LatsTaskEvent(Event):
    state: TreeState

class LatsSelectionEvent(LatsTaskEvent):
    pass

class LatsGenerationEvent(LatsTaskEvent):
    node: Node

class LatsFinalizationEvent(LatsTaskEvent):
    pass

class LatsAgentWorkflow(Workflow):
    def __init__(self, model, attempts, system_prompt,
hidden_rules=False, verbose=False, timeout=2400, **kwargs):
        super().__init__(timeout=timeout, **kwargs)
        self.model = model
```

```
        self.attempts = attempts
        self.branching_factor = 2
        self.system_prompt = system_prompt
        self.verbose = verbose

    async def snapshot(self, ctx: Context, state: TaskState):
        return state.conversation.copy()

    async def update_prompts_pile(self, ctx: Context,
conversation):
        all_prompts = await ctx.store.get("all_prompts", [])
        all_prompts.append(copy.deepcopy(conversation))
        await ctx.store.set("all_prompts", all_prompts)

    async def update_completions_pile(self, ctx: Context,
conversation):
        all_completions = await
ctx.store.get("all_completions", [])
        all_completions.append(copy.deepcopy(conversation))
        await ctx.store.set("all_completions", all_completions)

    async def generate(self, ctx: Context, state: TaskState):
        snapshot = await self.snapshot(ctx, state)
        state.agent_states.append(snapshot)

        agent_state = snapshot

        def role_mapping(role):
            if role == 'system':
                return MessageRole.SYSTEM
            elif role == 'user':
                return MessageRole.USER
            elif role == 'assistant':
                return MessageRole.ASSISTANT

        llamaindex_conversation = [
```

```
            ChatMessage(role=role_mapping(m['role']),
content=m['content']) for m in agent_state
        ]
        await self.update_prompts_pile(ctx, agent_state)

        response = await model.achat(llamaindex_conversation)
        message    =    {'role':    'assistant',    'content':
response.message.content}
        state.conversation.append(message)
        await self.update_completions_pile(ctx, [message])

        return state, message

    async def reflect(self, ctx: Context, state: TaskState):
        request = state.conversation[1]['content'] # Index of a
first user input
        answer = state.conversation[-2]['content']
        system_feedback = state.conversation[-1]['content']
        score = int(float(state.stats['score']) * 10)

        validator_prompt = f"""
You're helper supporting the assistants when they fail to provide
the correct solution to user requests.
You're only sent the solutions that are not 100% correct.

To help the assistant reflect to the assistant's response to the
user task.
Reflection should provide an insightful review for the assistant
and potential supervisors such as when attended too
the result should improve. To help you reason about the source
of the problems and their severity you will also be provided
with
a system feedback and system-assigned score.

Here is the example:
```

```
TASK: 'Task given to the assistant'

SOLUTION: 'Solution provided by the assistant'

SYSTEM FEEDBACK: 'Logs from an evaluator like a test suite or
the user'

SCORE: 'Score inferred from system feedback, often tied to
percent of passing tests'

REASONING: 'Your reasoning about giving the particular score'

Now, the actual task:

TASK:
{request}

SOLUTION:
{answer}

SYSTEM FEEDBACK:
{system_feedback}

SCORE: {score} / 10

REASONING:
        """

        agent_state = [
            {'role': 'user', 'content': validator_prompt},
            {'role': 'assistant', 'content': 'REASONING:'}
        ]

        llamaindex_conversation = [
            ChatMessage(**m) for m in agent_state
        ]
```

```
        response = await model.achat(llamaindex_conversation)
        await self.update_prompts_pile(ctx, [{'role': 'user',
'content': validator_prompt}])
        text = response.message.content
        reflections = text

        reflection = Reflection(
            reflections=reflections,
            score=score,
            found_solution=False
        )

        message = {"role": "user", "content": text}
        state.conversation.append(message)
        await self.update_completions_pile(ctx, [message])

        return reflection

    async def evaluate_and_reflect(self, ctx: Context, state:
TaskState):
        result = evaluate_task(state.task_data,
state.conversation, verbose=self.verbose)
        state.attempts += 1

        is_correct = result['is_correct']
        state.stats = {
            'pass@1': result.get('pass@1', 0),
            'score': result.get('score', 0),
            'best_score': max(state.stats.get('best_score', 0),
result.get('score', 0)),
            'correct': is_correct
        }

        if is_correct:
            return Reflection(
```

```
                reflections="Passes all tests",
                score=10,
                found_solution=True
            )
        else:
            message = {"role": "user", "content":
str(result['response'])}
            state.conversation.append(message)

            return await self.reflect(ctx, state)

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
    async def initial_step(self, ctx: Context, ev: StartEvent)
-> LatsSelectionEvent:
        state = ev.state
        message = ev.message

        state.conversation = [self.system_prompt, message]
        state.agent_states = []

        await ctx.store.set("agent_state",
state.conversation.copy())
        await ctx.store.set("attempts", 1)

        state, message = await self.generate(ctx, state)
        reflection = await self.evaluate_and_reflect(ctx,
state)

        root = Node(state, [message], reflection=reflection)
        tree_state = {
            "root": root,
            "input": message['content'],
        }

        return LatsSelectionEvent(state=tree_state)
```

```
    def select(self, root: Node) -> Node:
        """Starting from the root node a child node is selected
at each tree level until a leaf node is reached."""

        if not root.children:
            return root

        node = root
        while not node.is_terminal:
            max_child = max(node.children, key=lambda child:
child.upper_confidence_bound())
            node = max_child

        return node

    @step
    async def selection_step(self, ctx: Context, ev:
LatsSelectionEvent) -> LatsGenerationEvent |
LatsFinalizationEvent:
        state = ev.state

        root = state["root"]
        attempts = await ctx.store.get("attempts")

        if root.is_solved or attempts >= self.attempts:
            return LatsFinalizationEvent(state=state)

        await ctx.store.set("attempts", attempts + 1)
        return LatsGenerationEvent(state=state,
node=self.select(root))

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
```

```
    async def generation_step(self, ctx: Context, ev:
LatsGenerationEvent) -> LatsGenerationEvent |
LatsSelectionEvent:
        node = ev.node

        state = copy.deepcopy(node.state)
        state, message = await self.generate(ctx, state)
        reflection = await self.evaluate_and_reflect(ctx,
state)

        child = Node(state, [message], reflection=reflection,
parent=node)
        node.children.append(child)

        child_count = len(node.children)

        if child_count < self.branching_factor:
            return ev
        else:
            return LatsSelectionEvent(state=ev.state)

    @step
    async def finish_step(self, ctx: Context, ev:
LatsFinalizationEvent) -> StopEvent:
        best_state = ev.state['root'].get_best_solution().state
        best_state.attempts = await ctx.store.get("attempts")
        best_state.metadata['tree'] = ev.state

        all_prompts = await ctx.store.get("all_prompts", [])
        best_state.total_prompt_tokens =
recount_tokens(all_prompts)

        all_completions = await
ctx.store.get("all_completions", [])
        best_state.total_completion_tokens =
recount_tokens(all_completions)
```

```
        return StopEvent(result=best_state)
```

Лістинг А.4 Реалізація сценарію за методом SWE-Search

```
class SweSearchTaskEvent(Event):
    state: TreeState

class SweSearchSelectionEvent(SweSearchTaskEvent):
    pass

class SweSearchGenerationEvent(SweSearchTaskEvent):
    node: Node

class SweSearchFinalizationEvent(SweSearchTaskEvent):
    pass

class SweSearchAgentWorkflow(Workflow):
    def __init__(self, model, attempts, system_prompt,
verbose=False, timeout=1200, **kwargs):
        super().__init__(timeout=timeout, **kwargs)
        self.model = model
        self.attempts = attempts
        self.branching_factor = 2
        self.system_prompt = system_prompt
        self.verbose = verbose

    async def snapshot(self, ctx: Context, state: TaskState):
        return state.conversation.copy()

    async def update_prompts_pile(self, ctx: Context,
conversation):
        all_prompts = await ctx.store.get("all_prompts", [])
        all_prompts.append(copy.deepcopy(conversation))
        await ctx.store.set("all_prompts", all_prompts)
```

```
    async def update_completions_pile(self, ctx: Context, conversation):
        all_completions = await ctx.store.get("all_completions", [])
        all_completions.append(copy.deepcopy(conversation))
        await ctx.store.set("all_completions", all_completions)

    async def generate(self, ctx: Context, state: TaskState, suggestions: List[Reflection]=None):
        if suggestions is None:
            suggestions = []
        else:
            suggestions = [r.as_message() for r in suggestions]

        agent_state = []

        agent_state.append(state.conversation[0])
        agent_state.append(state.conversation[1])

        state.agent_states.append(agent_state)

        def role_mapping(role):
            if role == 'system':
                return MessageRole.SYSTEM
            elif role == 'user':
                return MessageRole.USER
            elif role == 'assistant':
                return MessageRole.ASSISTANT

        llamaindex_conversation = [
            ChatMessage(**m) for m in agent_state
        ]
        llamaindex_conversation.extend(suggestions)
        await self.update_prompts_pile(ctx, [{'role': m.role, 'content': m.content} for m in llamaindex_conversation])
```

```
        response = await model.achat(llamaindex_conversation)
        message    =    {'role':    'assistant',    'content':
response.message.content}
        state.conversation.append(message)
        await self.update_completions_pile(ctx, [message])

        return state, message

    async def reflect(self, ctx: Context, state: TaskState):
        request = state.conversation[1]['content'] # Index of a
first user input
        answer = state.conversation[-2]['content']
        system_feedback = state.conversation[-1]['content']
        score = int(float(state.stats['score']) * 10)

        validator_prompt = f"""
You're helper supporting the assistants when they fail to provide
the correct solution to user requests.
You're only sent the solutions that are not 100% correct.

To help the assistant reflect to the assistant's response to the
user task.
Reflection should provide an insightful review for the assistant
and potential supervisors such as when attended too
the result should improve. To help you reason about the source
of the problems and their severity you will also be provided
with
a system feedback and system-assigned score.

Here is the example:

TASK: 'Task given to the assistant'

SOLUTION: 'Solution provided by the assistant'
```

```
SYSTEM FEEDBACK: 'Logs from an evaluator like a test suite or
the user'

SCORE: 'Score inferred from system feedback, often tied to
percent of passing tests'

REASONING: 'Your reasoning about giving the particular score'

Now, the actual task:

TASK:
{request}

SOLUTION:
{answer}

SYSTEM FEEDBACK:
{system_feedback}

SCORE: {score} / 10

REASONING:
        """

        agent_state = [
            {'role': 'user', 'content': validator_prompt},
            {'role': 'assistant', 'content': 'REASONING:'}
        ]

        llamaindex_conversation = [
            ChatMessage(**m) for m in agent_state
        ]

        response = await model.achat(llamaindex_conversation)
        await  self.update_prompts_pile(ctx,  [{'role':  'user',
'content': validator_prompt}])
```

```
        text = response.message.content
        reflections = text

        reflection = Reflection(
            reflections=reflections,
            score=score,
            found_solution=False
        )

        message = {"role": "user", "content": text}
        state.conversation.append(message)
        await self.update_completions_pile(ctx, [message])

        return reflection

    async def evaluate_and_reflect(self, ctx: Context, state:
TaskState):
        result         =         evaluate_task(state.task_data,
state.conversation, verbose=self.verbose)
        state.attempts += 1

        is_correct = result['is_correct']
        state.stats = {
            'pass@1': result.get('pass@1', 0),
            'score': result.get('score', 0),
            'correct': is_correct
        }

        if is_correct:
            return Reflection(
                reflections="Passes all tests",
                score=10,
                found_solution=True
            )
        else:
```

```
            message = {"role": "user", "content":
str(result['response'])}
            state.conversation.append(message)

            return await self.reflect(ctx, state)

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
    async def initial_step(self, ctx: Context, ev: StartEvent)
-> SweSearchSelectionEvent:
        state = ev.state
        message = ev.message

        state.conversation = [self.system_prompt, message]
        state.agent_states = []

        await ctx.store.set("agent_state",
state.conversation.copy())
        await ctx.store.set("attempts", 1)

        state, message = await self.generate(ctx, state)
        reflection = await self.evaluate_and_reflect(ctx,
state)

        root = Node(state, [message], reflection=reflection)
        tree_state = {
            "root": root,
            "input": message['content'],
        }

        return SweSearchSelectionEvent(state=tree_state)

    def select(self, root: Node) -> Node:
        if not root.children:
            return root
```

```
        node = root
        while not node.is_terminal:
            max_child = max(node.children, key=lambda child:
child.upper_confidence_bound())
            node = max_child

        return node

    @step
    async def selection_step(self, ctx: Context, ev:
SweSearchSelectionEvent) -> SweSearchGenerationEvent |
SweSearchFinalizationEvent:
        state = ev.state
        root = state['root']

        attempts = await ctx.store.get("attempts")

        if root.is_solved or attempts >= self.attempts:
            return SweSearchFinalizationEvent(state=state)

        await ctx.store.set("attempts", attempts + 1)
        return SweSearchGenerationEvent(state=state,
node=self.select(root))

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
    async def generation_step(self, ctx: Context, ev:
SweSearchGenerationEvent) -> SweSearchSelectionEvent |
SweSearchGenerationEvent:
        node = ev.node

        state = copy.deepcopy(node.state)

        suggestions = []
        suggestions.append(node.reflection)
        for child in node.children:
```

```
            suggestions.append(child.reflection)

        state, message = await self.generate(ctx, state,
suggestions=suggestions)
        reflection = await self.evaluate_and_reflect(ctx,
state)

        child = Node(state, [message], reflection=reflection,
parent=node)
        node.children.append(child)

        child_count = len(node.children)

        if child_count < self.branching_factor:
            return ev
        else:
            return SweSearchSelectionEvent(state=ev.state)

    @step
    async def finish_step(self, ctx: Context, ev:
SweSearchFinalizationEvent) -> StopEvent:
        best_state = ev.state['root'].get_best_solution().state
        best_state.attempts = await ctx.store.get("attempts")
        best_state.metadata['tree'] = ev.state

        all_prompts = await ctx.store.get("all_prompts", [])
        best_state.total_prompt_tokens =
recount_tokens(all_prompts)

        all_completions = await
ctx.store.get("all_completions", [])
        best_state.total_completion_tokens =
recount_tokens(all_completions)

        return StopEvent(result=best_state)
```

Лістинг А.5 Реалізація сценарію за методом MCTS-Rewrite

```
class MctsRewriteTaskEvent(Event):
    state: TreeState

class MctsRewriteSelectionEvent(MctsRewriteTaskEvent):
    pass

class MctsRewriteRewriteEvent(MctsRewriteTaskEvent):
    node: Node
    feedback: list[str]

class MctsRewriteExpansionEvent(MctsRewriteTaskEvent):
    node: Node

class MctsRewriteGenerationEvent(MctsRewriteTaskEvent):
    node: Node
    node_state: Optional[TaskState] = None

class MctsRewriteFinalizationEvent(MctsRewriteTaskEvent):
    pass

class MctsRewriteAgentWorkflow(Workflow):
    def __init__(self, model, attempts, system_prompt,
verbose=False, timeout=1200, **kwargs):
        super().__init__(timeout=timeout, **kwargs)
        self.model = model
        self.attempts = attempts
        self.branching_factor = 2
        self.system_prompt = system_prompt
        self.verbose = verbose

    async def snapshot(self, ctx: Context, state: TaskState):
        return state.conversation.copy()
```

```
    async def update_prompts_pile(self, ctx: Context,
conversation):
        all_prompts = await ctx.store.get("all_prompts", [])
        all_prompts.append(copy.deepcopy(conversation))
        await ctx.store.set("all_prompts", all_prompts)

    async def update_completions_pile(self, ctx: Context,
conversation):
        all_completions = await
ctx.store.get("all_completions", [])
        all_completions.append(copy.deepcopy(conversation))
        await ctx.store.set("all_completions", all_completions)

    async def generate(self, ctx: Context, state: TaskState):
        agent_state = []

        agent_state.append(state.conversation[0])
        agent_state.append(state.conversation[1])

        state.agent_states.append(agent_state)

        def role_mapping(role):
            if role == 'system':
                return MessageRole.SYSTEM
            elif role == 'user':
                return MessageRole.USER
            elif role == 'assistant':
                return MessageRole.ASSISTANT

        llamaindex_conversation = [
            ChatMessage(**m) for m in agent_state
        ]

        response = await model.achat(llamaindex_conversation)
        message = {'role': 'assistant', 'content':
response.message.content}
```

```
        state.conversation.append(message)
        await self.update_prompts_pile(ctx, agent_state)
        await self.update_completions_pile(ctx, [message])

        return state, message

    async def rewrite(self, ctx: Context, state: TaskState, conflict_set: list[str]):
        request = state.conversation[1]['content']
        feedback = '\n'.join([f"* {tag}" for tag in conflict_set])

        rewrite_prompt = f"""
You are a helper for writing task descriptions. You will be given a current task description and a list of short summaries of the incorrect decisions assignees made when following the current version.

To maximize your helpfulness follow these rules:
* This will be the new task description, MENTIONING CURRENT/PREVIOUS SOLUTION AND OUTPUTS IS IRRELEVANT AND CONFUSING.
Prefer 'Solution will be tested on scale' or a target asymptotic time complexity over 'Previous solution was timing out'.
* Do not omit crucial details like function names or other conventions, etc if provided
* DELETE if the instruction VIOLATES THIS RULES, OTHERWISE REWRITE
* Follow the FEEDBACK over your judgement
* You're encouraged to provide examples (NOT SOLUTION PROPOSALS) as hints
* Provide relevant context, but don't try to be overwhelmingly verbose

TASK:
{request}
```

```
FALLACIES:
{feedback}
"""

        agent_state = [
            {'role': 'user', 'content': rewrite_prompt},
            {'role': 'assistant', 'content': 'UPDATED TASK:'}
        ]

        llamaindex_conversation = [
            ChatMessage(**m) for m in agent_state
        ]

        response = await model.achat(llamaindex_conversation)
        await self.update_prompts_pile(ctx, [{'role': 'user',
'content': rewrite_prompt}])
        new_task_message = {'role': 'assistant', 'content':
response.message.content}

        state = copy.deepcopy(state)
        state.conversation = [
            state.conversation[0],
            new_task_message
        ]
        await                self.update_completions_pile(ctx,
[new_task_message])

        return state

    async def reflect(self, ctx: Context, state: TaskState):
        request = state.conversation[1]['content']
        answer = state.conversation[-2]['content']
        system_feedback = state.conversation[-1]['content']
        score = int(float(state.stats['score']) * 10)
```

```
        validator_prompt = f"""
You're helper supporting the assistants when they fail to provide
the correct solution to user requests.
You're only sent the solutions that are not 100% correct.

To help the assistant reflect to the assistant's response to the
user task.
Reflection should provide an insightful review for the assistant
and potential supervisors such as when attended too
the result should improve. To help you reason about the source
of the problems and their severity you will also be provided
with
a system feedback and system-assigned score.

Here is the example:

TASK: 'Task given to the assistant'

SOLUTION: 'Solution provided by the assistant'

SYSTEM FEEDBACK: 'Logs from an evaluator like a test suite or
the user'

SCORE: 'Score inferred from system feedback, often tied to
percent of passing tests'

REASONING: 'Your reasoning about giving the particular score'

Now, the actual task:

TASK:
{request}

SOLUTION:
{answer}
```

```
SYSTEM FEEDBACK:
{system_feedback}

SCORE: {score} / 10

REASONING:
        """

        agent_state = [
            {'role': 'user', 'content': validator_prompt},
            {'role': 'assistant', 'content': 'REASONING:'}
        ]

        llamaindex_conversation = [
            ChatMessage(**m) for m in agent_state
        ]

        response = await model.achat(llamaindex_conversation)
        await self.update_prompts_pile(ctx, [{'role': 'user',
'content': validator_prompt}])
        text = response.message.content
        reflections = text

        reflection = Reflection(
            reflections=reflections,
            score=score,
            found_solution=False
        )

        message = {"role": "assistant", "content": text}
        state.conversation.append(message)
        await self.update_completions_pile(ctx, [message])

        return reflection
```

```
    async def evaluate_and_reflect(self, ctx: Context, state:
TaskState):
        result         =         evaluate_task(state.task_data,
state.conversation, verbose=self.verbose)
        state.attempts += 1

        is_correct = result['is_correct']
        state.stats = {
            'pass@1': result.get('pass@1', 0),
            'score': result.get('score', 0),
            'best_score': max(state.stats.get('best_score', 0),
result.get('score', 0)),
            'correct': is_correct
        }

        if is_correct:
            return Reflection(
                reflections="Passes all tests",
                score=10,
                found_solution=True
            )
        else:
            message    =    {"role":    "user",    "content":
str(result['response'])}
            state.conversation.append(message)

            return await self.reflect(ctx, state)

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
    async def initial_step(self, ctx: Context, ev: StartEvent)
-> MctsRewriteSelectionEvent:
        state = ev.state
        message = ev.message

        state.conversation = [self.system_prompt, message]
```

```
        state.agent_states = []

        await ctx.store.set("agent_state",
state.conversation.copy())
        await ctx.store.set("attempts", 1)

        state, message = await self.generate(ctx, state)
        reflection = await self.evaluate_and_reflect(ctx,
state)

        root = Node(state, [message], reflection=reflection)
        tree_state = {
            "root": root,
            "input": message['content'],
        }

        return MctsRewriteSelectionEvent(state=tree_state)

    def select(self, root: Node) -> Node:
        if not root.children:
            return root

        node = root
        while not node.is_terminal:
            max_child = max(node.children, key=lambda child:
child.upper_confidence_bound())
            node = max_child

        return node

    @step
    async def selection_step(self, ctx: Context, ev:
MctsRewriteSelectionEvent) -> MctsRewriteExpansionEvent |
MctsRewriteFinalizationEvent:
        state = ev.state
        root = state['root']
```

```
        attempts = await ctx.store.get("attempts")

        if root.is_solved or attempts >= self.attempts:
            return MctsRewriteFinalizationEvent(state=state)

        await ctx.store.set("attempts", attempts + 1)
        return           MctsRewriteExpansionEvent(state=state,
node=self.select(root))

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
    async    def    expand_step(self,    ctx:    Context,    ev:
MctsRewriteExpansionEvent)    ->    MctsRewriteRewriteEvent    |
MctsRewriteGenerationEvent:
        node = ev.node
        state = copy.deepcopy(node.state)

        suggestions = []
        suggestions.append(node.reflection.reflections)
        for child in node.children:
            suggestions.append(child.reflection.reflections)

        return          MctsRewriteRewriteEvent(state=ev.state,
node=ev.node, feedback=suggestions)

    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
    async    def    rewrite_step(self,    ctx:    Context,    ev:
MctsRewriteRewriteEvent) -> MctsRewriteGenerationEvent:
        state    =    await    self.rewrite(ctx,    ev.node.state,
ev.feedback)
        return       MctsRewriteGenerationEvent(state=ev.state,
node=ev.node, node_state=state)
```

```
    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=30,
max_attempts=5))
    async def generation_step(self, ctx: Context, ev:
MctsRewriteGenerationEvent) -> MctsRewriteSelectionEvent |
MctsRewriteExpansionEvent:
        node = ev.node
        state = node.state if ev.node_state is None else
ev.node_state

        state, message = await self.generate(ctx, state)
        reflection = await self.evaluate_and_reflect(ctx,
state)

        child = Node(state, [message], reflection=reflection,
parent=node)
        node.children.append(child)

        child_count = len(node.children)

        if child_count < self.branching_factor:
            return MctsRewriteExpansionEvent(state=ev.state,
node=ev.node)
        else:
            return MctsRewriteSelectionEvent(state=ev.state)

    @step
    async def finish_step(self, ctx: Context, ev:
MctsRewriteFinalizationEvent) -> StopEvent:
        best_state = ev.state['root'].get_best_solution().state
        best_state.attempts = await ctx.store.get("attempts")

        all_prompts = await ctx.store.get("all_prompts", [])
        best_state.total_prompt_tokens =
recount_tokens(all_prompts)
        all_completions = await
ctx.store.get("all_completions", [])
```

```
        best_state.total_completion_tokens                    =
recount_tokens(all_completions)

        best_state.metadata['all_prompts']                    =
copy.deepcopy(all_prompts)
        best_state.metadata['all_completions']                =
copy.deepcopy(all_completions)

        return StopEvent(result=best_state)
```

Лістинг А.6 Реалізація сценарію за методом LLM-CSP

```
class LlmCspTaskEvent(Event):
    state: TaskState

class LlmCspLabelingEvent(LlmCspTaskEvent):
    pass

class LlmCspUnlabelingEvent(LlmCspTaskEvent):
    pass

class LlmCspCspEvent(LlmCspTaskEvent):
    variables: list[str] = []
    constraints: list[str] = []
    conflict_set: list[str] = []

class LlmCspReflectEvent(LlmCspCspEvent):
    pass

class LlmCspAnalyzeEvent(LlmCspCspEvent):
    pass

class LlmCspCheckEvent(LlmCspCspEvent):
    pass
```

```
class LlmCspRewriteEvent(LlmCspCspEvent):
    pass


class LlmCspFinalizationEvent(LlmCspTaskEvent):
    pass


class LlmCspAgentWorkflow(Workflow):
    def __init__(self, model, attempts, system_prompt,
verbose=False, timeout=900, **kwargs):
        super().__init__(timeout=timeout, **kwargs)
        self.model = model
        self.attempts = attempts
        self.system_prompt = system_prompt
        self.constraints_cache = []
        self.verbose = verbose


    def extract_list_items(self, text, deduplicate=False):
        pattern = r"^[\s\t]*[•\-\*+]\s+(.*)$" # r"[*] (.*)"


        list_items = []
        for match in re.finditer(pattern, text,
flags=re.MULTILINE):
            captured_text = match.group(1).strip()


            if captured_text == "":
                continue


            list_items.append(captured_text)


        if deduplicate:
            list_items = list(set(list_items))
```

```
        return list_items


    async def snapshot(self, ctx: Context, state: TaskState):
        return state.conversation.copy()


    async def update_prompts_pile(self, ctx: Context,
conversation):
        all_prompts = await ctx.store.get("all_prompts", [])
        all_prompts.append(copy.deepcopy(conversation))
        await ctx.store.set("all_prompts", all_prompts)


    async def update_completions_pile(self, ctx: Context,
conversation):
        all_completions = await
ctx.store.get("all_completions", [])
        all_completions.append(copy.deepcopy(conversation))
        await ctx.store.set("all_completions",
all_completions)


    async def rewrite(self, ctx: Context, state: TaskState,
conflict_set: list[str]):
        request = state.conversation[1]['content']


        if len(conflict_set) > 0:
            feedback = '\n'.join([f"* {tag}" for tag in
conflict_set])
            rewrite_prompt = f"""
You are a helper for writing task descriptions. You will be
given a current task description and a list of short summaries
of the
incorrect decisions assignees made when following the current
version.


To maximize your helpfulness follow these rules:
* Your submission won't be further processed, so do not format
it as an answer and don't add any metadata
```

* Prefer boundaries over suggestions DO NOT X > DO Y. Although contrastive will also do: DO NOT X, BETTER DO Y.

* This will be the new task description, MENTIONING CURRENT/PREVIOUS SOLUTION AND OUTPUTS IS IRRELEVANT AND CONFUSING.

Prefer 'Avoid simple, but slow solutions' over 'Previous solution was timing out'.

* DO NOT OMIT FUNCTION NAMES, EXPECTED INPUTS AND OUTPUTS or other conventions, etc if provided, preserve function interface examples

* DELETE if the instruction VIOLATES THIS RULES, OTHERWISE REWRITE

* Follow the FEEDBACK over your judgement

* You're encouraged to provide hints to make the task more direct, BUT DO NOT WRITE THE EXAMPLE SOLUTION, KEEP JUST THE INTERFACE

* Provide relevant context, but BE CONCISE

FALLACIES:

{feedback}

TASK:

{request}

"""

        else:

            rewrite_prompt = f"""

You are a helper for writing task descriptions. We want to have our descriptions made more diverse, so there are different

variants of them encouraging the assignees to be more creative.

You will be given some existing description. Your job is to create another one that preserves relevant context, but is

written differently. You may try introducing hints, advices or examples through analogies. Also, please,

```
do not omit crucial details like function names or other
conventions, etc if provided.

Your submission won't be further processed, so do not format
it as an answer and don't add any metadata.


TASK:

{request}

"""

        agent_state = [

            {'role': 'user', 'content': rewrite_prompt},

            {'role': 'assistant', 'content': 'UPDATED TASK:'}

        ]


        llamaindex_conversation = [

            ChatMessage(**m) for m in agent_state

        ]


        response = await model.achat(llamaindex_conversation)

        await self.update_prompts_pile(ctx, [{'role': 'user',
'content': rewrite_prompt}])


        new_task_message = {'role': 'user', 'content':
response.message.content}


        state = copy.deepcopy(state)

        state.conversation.append(rewrite_prompt)

        state.conversation.append(new_task_message)

        await self.update_completions_pile(ctx,
[new_task_message])


        return state


    async def generate(self, ctx: Context, state: TaskState):

        agent_state = []
```

```
        agent_state.append(state.conversation[0])
        agent_state.append(state.conversation[-1])

        state.agent_states.append(agent_state)

        def role_mapping(role):
            if role == 'system':
                return MessageRole.SYSTEM
            elif role == 'user':
                return MessageRole.USER
            elif role == 'assistant':
                return MessageRole.ASSISTANT

        llamaindex_conversation = [
            ChatMessage(role=role_mapping(m['role']),
content=m['content']) for m in agent_state
        ]
        await self.update_prompts_pile(ctx, agent_state)

        response = await
self.model.achat(llamaindex_conversation)

        message = {'role': 'assistant', 'content':
response.message.content}
        state.conversation.append(message)
        await self.update_completions_pile(ctx, [message])

        return state, message

    async def analyze(self, ctx: Context, state: TaskState):
        system_prompt = state.conversation[0]
        request = state.conversation[-3]
```

```
        answer = state.conversation[-2]
        system_feedback = state.conversation[-1]

        agent_state = [
            system_prompt,
            request,
            answer,
            system_feedback
        ]

        analyzer_prompt = f"""
Rewrite the feedback (which is a score from 0 to 1
corresponding to tests covered and data about failed test
case) in natural langauage and then analyze the case.

Summarize the constraints that are evident from the task and
provided feedback to the proposed solution.

Analyze deeply: most of the tests are not corner cases, you
need to provide an actual constraint because the possible

explicit constraints can be infinite.

Be concise: constraints should not include duplicates or have
the constraints for which cases are already covered by another
constraint.

Remember, that if there is an error you should define the
constraint, not just report that there is an error. If you
can't figure it now, simply

write it as a 'Input X should return Y'


Summary must look like this:

* Function name should be `get_min_value`

* If the input is empty return None

* Should be faster than O(n^2)

...

* Return the first min value
        """
```

```
        def role_mapping(role):
            if role == 'system':
                return MessageRole.SYSTEM
            elif role == 'user':
                return MessageRole.USER
            elif role == 'assistant':
                return MessageRole.ASSISTANT


        llamaindex_conversation = [
            ChatMessage(role=role_mapping(m['role']),
content=m['content']) for m in agent_state
        ]


        analyzer_message = {'role': 'user', 'content':
analyzer_prompt}

llamaindex_conversation.append(ChatMessage(**analyzer_message)
)
        await self.update_prompts_pile(ctx, [{'role': m.role,
'content': m.content} for m in llamaindex_conversation])


        response = await
self.model.achat(llamaindex_conversation)
        response_message = {'role': response.message.role,
'content': response.message.content}
        state.conversation.append(analyzer_message)
        state.conversation.append(response_message)


        await self.update_completions_pile(ctx,
[response_message])


        analysis =
self.extract_list_items(response.message.content,
deduplicate=True)
        return analysis
```

```
    async def reflect(self, ctx: Context, state: TaskState):
        system_prompt = state.conversation[0]
        request = state.conversation[-7]
        answer = state.conversation[-6]

        agent_state = [
            system_prompt,
            request,
            answer
        ]

        reflector_prompt = f"""
Now, summarize your thinking process as a list of statements.
If you think there were multiple reasons to your decision -
provide both.
But, don't be too verbose and exclude the decisions that are
already covered like "I need to use nested loops" then "I
should loop over xs".

Like this:
* Function name is `get_min_value`
* If the input is empty return None
* Inspect the elements recursively
* Use the nested list
...
* Return the min value

Now, your summary
        """

        def role_mapping(role):
            if role == 'system':
                return MessageRole.SYSTEM
```

```
            elif role == 'user':
                return MessageRole.USER
            elif role == 'assistant':
                return MessageRole.ASSISTANT


        llamaindex_conversation = [
            ChatMessage(role=role_mapping(m['role']),
content=m['content']) for m in agent_state
        ]


        reflector_message = {'role': 'user', 'content':
reflector_prompt}

llamaindex_conversation.append(ChatMessage(**reflector_message
))
        await self.update_prompts_pile(ctx, [{'role': m.role,
'content': m.content} for m in llamaindex_conversation])


        response = await
self.model.achat(llamaindex_conversation)
        response_message = {'role': response.message.role,
'content': response.message.content}
        state.conversation.append(reflector_message)
        state.conversation.append(response_message)


        await self.update_completions_pile(ctx,
[response_message])


        reflection =
self.extract_list_items(response.message.content,
deduplicate=True)
        return reflection


    async def reduce_cache(self, ctx: Context, state,
constraints: list[str]):
        reduce_prompt = f"""
```

```
You will be given the list of constraints inferred from the
prompt and system feedback. Your task is to optimize this list
by removing the

constraints that are duplicate, redundant or dominated by
stronger constraints provided. You should approach
deduplication smartly by selecting

the most clear condition as the one to keep.


Constraints:

{'\n'.join([f"* {c}" for c in constraints])}



"""

        reduce_message = {'role': 'user', 'content':
reduce_prompt}

        agent_state = [

            {'role': 'user', 'content': reduce_prompt},

            {'role': 'assistant', 'content': 'Improved
constraints:'}

        ]


        llamaindex_conversation = [

            ChatMessage(**m) for m in agent_state

        ]


        response = await model.achat(llamaindex_conversation)

        await self.update_prompts_pile(ctx, [reduce_message])


        constraints_str = response.message.content

        constraints_message = {'role': 'assistant', 'content':
constraints_str}


        state.conversation.append(reduce_message)

        state.conversation.append(constraints_message)
```

```
        await self.update_completions_pile(ctx,
[constraints_message])


        list_items = self.extract_list_items(constraints_str)


        return list_items


    async def llm_check(self, ctx: Context, state,
constraints: list[str], variables: list[str]):

        feedback = state.conversation[-7]['content']

        constraints_list = '\n'.join([f"* {c}" for c in
constraints])

        variables_list = '\n'.join([f"* {v}" for v in
variables])


        check_prompt = f"""

You are semantic consistency checker. You will be given a set
of constraints and a set of variables

as well as a system feedback to the solution which is a score
(from 0 to 1) corresponding to tests covered and data about
failed test case.

Provide a brief analysis on the relationships (independent, in
conflict, agrees etc) between constraints and assigned
variables

(variables can also conflict with each other). Then you need
to produce a "conflict set" that will contain descriptions of
existing conflicts

based on conflicting constraints or variables. It is better if
you do conflict-directed jumps omitting the conflicts that
exist only due to other conflicts.

Provided conflicts should not be suggestive: Prefer 'formula f
is incorrect' over 'formula f is incorrect, g should be used'.

Please, return the data in the same format of a bullet-point
list as the parser will consider any bullet-point line a
conflict (meaning you shouldn't use bullets in your analysis).


CONSTRAINTS:

{constraints_list}
```

```
VARIABLES:
{variables_list}

FEEDBACK:
{feedback}

"""

        check_message = {'role': 'user', 'content':
check_prompt}
        agent_state = [
            {'role': 'user', 'content': check_prompt},
            {'role': 'assistant', 'content': 'ANALYSIS:'}
        ]

        llamaindex_conversation = [
            ChatMessage(**m) for m in agent_state
        ]

        response = await model.achat(llamaindex_conversation)
        await self.update_prompts_pile(ctx, [{'role': 'user',
'content': check_prompt}])
        await self.update_prompts_pile(ctx, [check_message])

        conflicts_str = response.message.content
        conflicts_message = {'role': 'assistant', 'content':
conflicts_str}

        state.conversation.append(check_message)
        state.conversation.append(conflicts_message)
        await self.update_completions_pile(ctx,
[conflicts_message])
```

```
        list_items = self.extract_list_items(conflicts_str)
        return list_items


    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=20,
max_attempts=10))
    async def initial_step(self, ctx: Context, ev: StartEvent)
-> LlmCspLabelingEvent:
        state = ev.state
        message = ev.message

        state.conversation = [self.system_prompt, message]
        state.agent_states = []

        state.conversation.append(message)
        await ctx.store.set("attempts", 0)

        return LlmCspLabelingEvent(state=state)


    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=20,
max_attempts=10))
    async def label_step(self, ctx: Context, ev:
LlmCspLabelingEvent) -> LlmCspUnlabelingEvent |
LlmCspFinalizationEvent:
        state = ev.state
        attempts = await ctx.store.get("attempts")

        if attempts >= self.attempts:
            return LlmCspFinalizationEvent(state=state)

        state, message = await self.generate(ctx, state)

        await ctx.store.set("attempts", attempts + 1)
        state.attempts += 1
```

```
        return LlmCspUnlabelingEvent(state=state)


    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=20,
max_attempts=10))
    async def analyze_step(self, ctx: Context, ev:
LlmCspAnalyzeEvent) -> LlmCspReflectEvent:
        state = ev.state


        constraints = await self.analyze(ctx, state)
        self.constraints_cache = await self.reduce_cache(ctx,
state, self.constraints_cache + constraints)
        ev.constraints = self.constraints_cache


        return LlmCspReflectEvent(state=ev.state,
constraints=ev.constraints)


    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=20,
max_attempts=10))
    async def check_step(self, ctx: Context, ev:
LlmCspCheckEvent) -> LlmCspLabelingEvent | LlmCspRewriteEvent:
        state = ev.state


        conflict_set = await self.llm_check(ctx, state,
ev.constraints, ev.variables)


        if len(conflict_set) > 0:
            return LlmCspRewriteEvent(state=ev.state,
conflict_set=conflict_set)
        else:
            state = copy.deepcopy(state)
            return LlmCspLabelingEvent(state=state)


    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=20,
max_attempts=10))
```

```
    async def reflect_step(self, ctx: Context, ev:
LlmCspReflectEvent) -> LlmCspLabelingEvent | LlmCspCheckEvent
| LlmCspRewriteEvent:

        state = ev.state

        ev.variables = await self.reflect(ctx, state)

        return LlmCspCheckEvent(state=ev.state,
constraints=ev.constraints, variables=ev.variables)


    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=20,
max_attempts=10))

    async def rewrite_step(self, ctx: Context, ev:
LlmCspRewriteEvent) -> LlmCspLabelingEvent:

        state = await self.rewrite(ctx, ev.state,
ev.conflict_set)

        return LlmCspLabelingEvent(state=state)


    @step(retry_policy=GoogleGenAIResiliencePolicy(delay=20,
max_attempts=10))

    async def check_and_unlabel_step(self, ctx: Context, ev:
LlmCspUnlabelingEvent) -> LlmCspAnalyzeEvent |
LlmCspFinalizationEvent:

        state = ev.state

        result = evaluate_task(state.task_data,
state.conversation, verbose=self.verbose)


        is_correct = result['is_correct']

        state.stats = {

            'pass@1': result.get('pass@1', 0),

            'score': result.get('score', 0),

            'best_score': max(state.stats.get('best_score',
0), result.get('score', 0)),

            'correct': is_correct

        }


        if is_correct:

            return LlmCspFinalizationEvent(state=state)
```

```
        message = {"role": "user", "content": f"Score:
{result.get('score', 0)} Response: {str(result['response'])}"}
        state.conversation.append(message)


        return LlmCspAnalyzeEvent(state=state)



    @step
    async def finish_step(self, ctx: Context, ev:
LlmCspFinalizationEvent) -> StopEvent:
        state = ev.state


        all_prompts = await ctx.store.get("all_prompts", [])
        state.total_prompt_tokens =
recount_tokens(all_prompts)


        all_completions = await
ctx.store.get("all_completions", [])
        state.total_completion_tokens =
recount_tokens(all_completions)


        return StopEvent(result=state)
```

ДОДАТОК Б

Вихідний код для методів на основі семплювання

Лістинг Б.1 Реалізація методу Entropy-bruteforce

```
def entropy_based_sampler(model, messages, dsu, metric='score',
threshold=0.4,          l=-1,          delay_registration=True,
generation_kwargs=None):
    if isinstance(messages, str):
        messages = [{"role": "user", "content": messages}]

    generation_prompt                                          =
model.tokenizer.apply_chat_template(messages,    tokenize=True,
return_tensors='pt', add_generation_prompt=True)

    intercept_layer = model.model.layers[l]
    entropies = []
    varentropies = []
    scores = []
    lens_tokens = []
    tokens = []

    if delay_registration:
        queue = []

    hit_threshold = [False]
    ids = []
    activation_norm = [None]

    if generation_kwargs is None:
        generation_kwargs      =      dict(max_new_tokens=1024,
do_sample=False)

    with model.generate(generation_prompt, **generation_kwargs)
as tracer:
        with tracer.iter[:] as iterator:
```

```
            if iterator > 0:
                input_tokens                                    =
model.model.embed_tokens.input[0]
                token = input_tokens[-1].item().save()
                tokens.append(token)

            if hit_threshold[0]:
                if delay_registration:
                    queue.append((activation_norm[0], token))
                else:
                    if ids is None:
                        ids = []

                    dsu[activation_norm[0]] = ids + [token]

            layer_output = intercept_layer.output[0]
            activation = layer_output[-1].detach()#.save()
            activation_norm[0]      =      (activation      /
activation.norm(p=2))

            all_logits                                      =
model.lm_head(model.model.norm(layer_output))
            logits = all_logits[-1].unsqueeze(0).save()
            probs = torch.nn.functional.softmax(logits, dim=-1)
            log_probs = torch.nn.functional.log_softmax(logits,
dim=-1)
            entropy  =  (-torch.sum(probs  *  log_probs,  dim=-
1)).detach()
            scaled_entropy = entropy / np.log(logit_dim)
            entropies.extend(scaled_entropy.cpu().tolist())

            varentropy  =  torch.sum(probs  *  (log_probs  +
entropy.unsqueeze(-1))**2, dim=-1)
            scaled_varentropy    =    4    *    varentropy    /
np.log(logit_dim)**2
```

```
varentropies.extend(scaled_varentropy.cpu().tolist())

            score      =      torch.sqrt(scaled_entropy      *
scaled_varentropy)
            scores.extend(score.cpu().tolist())

            if metric == 'score':
                metric_value = score.item()
            elif metric == 'entropy':
                metric_value = scaled_entropy.item()
            elif metric == 'varentropy':
                metric_value = scaled_varentropy.item()
            else:
                raise ValueError("Unknown metric")

            hit_threshold[0] = (scaled_entropy > threshold and
scaled_varentropy > threshold).item()

            weights = torch.ones_like(model.lm_head.output[0, -
1])
            if hit_threshold[0]:
                ids = dsu[activation_norm[0]]
                if ids is not None and len(ids) > 0:
                    # print(weights.shape, ids)
                    used_ids = [x for x in ids]
                    used_ids.reverse()
                    used_ids = list(dict.fromkeys(used_ids))
                    w = [1 - np.exp(-i * np.log(2) / 5) for i in
range(len(used_ids))]

                    weights[torch.tensor(used_ids)]           =
torch.tensor(w, dtype=weights.dtype).to(weights.device)

            lens_tokens.append(logits[0].argmax(dim=-1))
```

```
            model.lm_head.output[0,            -1]            =
model.lm_head.output[0, -1] * weights

        token = tokens[-1]
        if hit_threshold[0]:
            if delay_registration:
                queue.append((activation_norm[0], token))
            else:
                if ids is None:
                    ids = []

                dsu[activation_norm[0]] = ids + [token]

    if delay_registration:
        for activation_norm, token in queue:
            ids = dsu[activation_norm]

            if ids is None:
                ids = []

            dsu[activation_norm] = ids + [token]

    return model.tokenizer.decode(tokens), dsu
```

Лістинг Б.2 Реалізація методів Entropy-search та Entropy-CSP

```
def entropy_search_based_sampler(
    model, messages, context, metric='score', threshold=0.3, l=-
1,            delay_registration=True,            verbose=True,
use_reward_penalty=True, generation_kwargs=None
):
    dsu = context['dsu']
    root = context['root']
    threshold = context['threshold']

    if isinstance(messages, str):
```

```
        messages = [{"role": "user", "content": messages}]

    generation_prompt                                          =
model.tokenizer.apply_chat_template(messages,    tokenize=True,
return_tensors='pt', add_generation_prompt=True)

    intercept_layer = model.model.layers[l]
    entropies = []
    varentropies = []
    scores = []
    lens_tokens = []
    tokens = []

    if delay_registration:
        queue = []

    hit_threshold = [False]
    node = None
    nodes = [root]
    activation_norm = [None]

    proxy_token = None

    def register_node(node, activation, token):
        nonlocal proxy_token

        if node is None:
            node = Node()

        dsu[activation] = node
        nodes[-1].add_child(node, proxy_token)
        nodes.append(node)

        proxy_token = token

    def apply_penalty(node, model_probs, logits):
```

```
        penalties = torch.zeros_like(logits)
        magnitude = torch.max(logits) + 1e-6

        sample_temperature = 3.0

        probs     =     torch.nn.functional.softmax(logits     /
sample_temperature, dim=-1)

        if verbose:
        if (len(node.children)) == 0:
            return penalties, []

        _, indices = torch.topk(probs, 32)

        indices       =       list(set(indices.tolist()       +
list(node.actions.keys())))
        top_probs = probs[indices].tolist()

        unexplored_top_probs = 0
        probs_mapping = {}
        for p, i in zip(top_probs, indices):
            probs_mapping[i] = p

            if i in node.actions:
                continue

            unexplored_top_probs += p

        action_pucbs = {}
        for action, children in node.actions.items():
            pucbs = [
                node.children[c].upper_confidence_bound(
                    node.children_visits[action][c],
use_reward_penalty=use_reward_penalty
                ) / (node.children_visits[action][c] + 1) for c
in children
```

```
            ]
            all_visits                                          =
sum(node.children_visits[action].values())

            action_pucbs[action] = sum(pucbs) / all_visits *
probs_mapping[action]

        best_action_id          =          max(action_pucbs,
key=action_pucbs.get)
        best_action_pucb = action_pucbs[best_action_id]
        exploration_pucb     =     unexplored_top_probs     *
math.sqrt(node.visits) / (1 + node.visits)
        exploration = best_action_pucb < exploration_pucb

        penalty_ids = [a for a in node.actions.keys()]

        if not exploration:
            penalty_ids.remove(best_action_id)

        w = [magnitude for i in range(len(penalty_ids))]

        if len(penalty_ids) == 0:
            return penalties, []

        penalties[torch.tensor(penalty_ids)] = torch.tensor(w,
dtype=penalties.dtype).to(penalties.device)
        return penalties, penalty_ids

    if generation_kwargs is None:
        generation_kwargs     =     dict(max_new_tokens=1024,
do_sample=False)

    with model.generate(generation_prompt, **generation_kwargs)
as tracer:
        with tracer.iter[:] as iterator:
            if iterator > 0:
```

```
                input_tokens                                    =
model.model.embed_tokens.input[0]
                token = input_tokens[-1].detach().item().save()
                tokens.append(token)

                queue.append((activation_norm[0], token))

            layer_output = intercept_layer.output[0]
            activation = layer_output[-1].detach() #.save()
            activation_norm[0]      =      (activation      /
activation.norm(p=2))

            all_logits                                          =
model.lm_head(model.model.norm(layer_output))
            logits                =                all_logits[-
1].detach().unsqueeze(0).save()
            probs = torch.nn.functional.softmax(logits, dim=-1)
            log_probs = torch.nn.functional.log_softmax(logits,
dim=-1)
            entropy = (-torch.sum(probs * log_probs, dim=-1))
            scaled_entropy = entropy / np.log(logit_dim)
            entropies.extend(scaled_entropy.cpu().tolist())

            varentropy   =   torch.sum(probs   *   (log_probs   +
entropy.unsqueeze(-1))**2, dim=-1)
            scaled_varentropy    =    4    *    varentropy    /
np.log(logit_dim)**2

varentropies.extend(scaled_varentropy.cpu().tolist())

            hit_threshold[0] = (scaled_entropy > threshold and
scaled_varentropy > threshold).item()

            lm_output = model.lm_head.output[0, -1].detach()
            penalties = torch.zeros_like(lm_output)
```

```
            p_ids = []
            if hit_threshold[0]:
                node = dsu[activation_norm[0]]
                if node is not None:
                    probs                                        =
torch.nn.functional.softmax(lm_output / 1.0, dim=-1)
                    penalties,   p_ids   =   apply_penalty(node,
probs, lm_output)

            lens_tokens.append(logits[0].argmax(dim=-1))
            model.lm_head.output[0,            -1]            =
model.lm_head.output[0, -1] - penalties

        token = tokens[-1]
        queue.append((activation_norm[0], token))

    for activation_norm, token in queue:
        node = dsu[activation_norm]
        register_node(node, activation_norm, token)

    context['entropies'] = (entropies + context['entropies'])[-
10000:]
    context['varentropies']        =        (entropies        +
context['varentropies'])[-10000:]

    return model.tokenizer.decode(tokens), context, nodes
```

ДОДАТОК В

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	130 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Стрельцов О.С.			Вдосконалення агентів LLM шляхом їх використання в якості пошукових агентів	Шифр групи	Код напр./спец.
Перевірив		Вітько О.В.				СШІм-24-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Чала Л.Е.					