

Wprowadzenie do przetwarzania języka naturalnego – Politechnika Warszawska

CoAct – Dokumentacja końcowa projektu Systemu Wieloagentowego do realizacji zadań programistycznych

Dokumentacja końcowa projektu CoAct – Systemu wieloagentowego działającego w obszarze Inżynierii Oprogramowania. Dokument ma na celu przedstawić podstawowe założenia projektu oraz jego stos technologiczny.

Cezary Dymicki, Jan Konarski, Gabriela Czarnowicz
2025-11-11

Spis treści

Spis Ilustracji	1
Definicja problemu	3
Studia literaturowe	4
Zapoznanie z literaturą	4
Baza Wiedzy Aplikacji	6
Spis modeli i ich rozmiarów:	8
Stos technologiczny projektu:	8
Tabela modeli językowych:.....	9
Strategia realizacji zadania	10
Strategia ewaluacji wyników	12
Wyniki wstępne z przeprowadzonych testów	14
Eksperyment.....	19
CoAct RAG	20
Analiza testów ewaluacyjnych systemu	21
Porównanie wyników z przykładem w literaturze	27
Instrukcja obsługi i instalacji systemu CoAct.....	29
Wnioski i podsumowanie.....	31

Spis Ilustracji

Rysunek 1 - Spis dostępnych baz wektorowych do wdrożenia systemu RAG.	6
Rysunek 2 - Przykładowa implementacja podobnego systemu, MetaGPT	7
Rysunek 3 - Przykładowa rola w zespole, Clarifier. Opis zachowania oraz celu pracy agenta.	11
Rysunek 4 - Przykładowe zadania testowe dla agentów.	13
Rysunek 5 - Początek realizacji zadania, projekt Kalkulatora zlecony zespołowi.	14
Rysunek 6 - Fragment wygenerowanego kodu przez agenta Programistę.....	15
Rysunek 7 - Przykład komunikacji między agentami, oraz sprawdzenie spełnienia wymagań zadania.	15

Rysunek 8 - Analiza wygenerowanego kodu przez agenta specjalizującego się w podatnościami ujawnia potencjalne wektory ataku na aplikację. W tym celu agent proponuje wprowadzenie poprawek w kodzie przekazywanym klientowi.	16
Rysunek 9 - Przykładowo wygenerowany kalkulator, jako wynik działania systemu wieloagentowego.	17
Rysunek 10 - Przykładowo wygenerowana gra w Snake'a, jako drugie zadanie zlecone systemu wieloagentowemu w trybie wykonywania sekwencyjnego.	17
Rysunek 11 - Przykładowe uruchomienie programu z terminalu IDE.	18
Rysunek 12 - Wygenerowany skrypt do animacji obrotu kostki według globalnej osi Y.	19
Rysunek 13 - Przykład prostej heurystyki wyszukiwania dokumentów o konkretnej tematyce.	20
Rysunek 14 - Przykład ewaluacji systemu przy pomocy zespołu agentów.	21
Rysunek 15 - Mechanizm punktacji wygenerowanego kodu.	22
Rysunek 16 - Ilustracja działania metryki G-Eval z oficjalnej strony DeepEval.	23
Rysunek 17 - Przykład implementacji metryki GEval w systemie CoAct. Readibility jest testem ogólnym.	24
Rysunek 18 - Pierwsze wyniki ewaluacji przy pomocy metryki GEval.	24
Rysunek 19 - Ewaluacja przy pomocy heurystyki wieloagentowej zwróciła wynik około 58% zgodności z wymaganiami użytkownika i zleonymi taskami.	25
Rysunek 20 - Głęboka ewaluacja dla tego samego kodu zwraca 80% zgodności z wymaganiami użytkownika. Doprecyzowanie: Readibility uwzględnia przestrzeganie dobrych praktyk programowania, redukcję duplikatów i czytelność kodu z perspektywy użytkownika lub programisty.	25
Rysunek 21 - Wizualizacja procesu ewaluacyjnego w systemie MAJ-EVAL, który również wykorzystuje debatę agentów do oceny swojej pracy. [1]....	27
Rysunek 22 - Wizualizacja procesów Individual-Agent-as-a-Judge oraz Multi-Agent Debate.	28
Rysunek 23 - Porównanie wyników do danych uzyskanych w poprzednio wspomnianych badaniach.	28
Rysunek 24 - Lista dostępnych opcji uruchamiania systemu CoAct.	29
Rysunek 25 - Przykład obecnie wykorzystywanych LLM'ów w pliku llm_providers.yaml w konfiguracji.	30
Rysunek 26 - Inne opcje dla providerów modeli językowych zapewnianych w ramach integracji przez bazowy framework od CrewAI. Dostępne są między innymi LLM'y od: OpenAI, Gemini, Azure, Anthropic.	30

Definicja problemu

Głównym problemem w dziedzinie inżynierii oprogramowania są wysokie nakłady finansowe jak i czasowe, które są wymagane do realizacji przedsięwzięcia informatycznego. Ponadto, niedokładności w estymacji zakresu pracy mogą spowodować opóźnienia w pracy, przekładające się bezpośrednio na przekroczenie pierwotnie wyznaczonego budżetu projektu.

Aby rozwiązać ten problem, do produkcji oprogramowania możliwe jest wykorzystanie agentów działających na silniku dużych modeli językowych (LLM – Large Language Models). Programy takie są w stanie współpracować z człowiekiem (np. deweloperem albo samym klientem), dzięki czemu możliwe jest ułatwienie procesu wytwarzania oprogramowania, ale i także uproszczenia procesu planowania kolejnych etapów pracy w zależności od przyjętego podejścia. Problem, jednakże przejawia się w braku kompetencji pojedynczego modelu językowego do pełnienia wszystkich funkcji w zespole symultanicznie. Agent często ma problemy ze szczegółami, może generować błędy lub wymaga wielu edycji i interwencji od samego użytkownika, co może zaburzyć tok pracy w firmie lub organizacji.

Wykorzystując kilku agentów jednocześnie jesteśmy w stanie zminimalizować ten problem, wykorzystując mechanizmy samo-uwagi modeli językowych, oraz wcielania się w role (role-playing), możemy przystosować LLM do wykonywania konkretnych zadań z danej dziedziny, dla przykładu pełnienia roli krytyka wygenerowanego kodu. Dzięki temu jesteśmy w stanie zminimalizować ryzyko na halucynację modeli językowych, do czego mają tendencję, jeśli nie są nadzorowane. Do realizacji powyższego projektu wykorzystamy zespół agentów. Zostanie o nich wspomniane w rozdziale – **Strategia realizacji zadania**.

Studia literaturowe

1. Road Ahead Junda He, Christoph Treude, David Lo, 2024, **LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead** - <https://arxiv.org/html/2404.04834v3>
 2. Vali Tawosi, Keshav Ramani, Salwa Alimir, Xiaomo Liu, 2025, **ALMAS: an Autonomous LLM-based Multi-Agent Software Engineering Framework** - <https://arxiv.org/pdf/2510.03463>
 3. Jiaju Chen, Yuxuan Lu, Xiaojie Wang, et. al. 2025 - **Multi-Agent-as-Judge: Aligning LLM-Agent-Based Automated Evaluation with Multi-Dimensional Human Evaluation** - <https://arxiv.org/abs/2507.21028>
-

Zapoznanie z literaturą

Role w tym projekcie zostały wybrane na podstawie podobnych projektów takich jak: MetaGPT, AgileCoder, ChatDev, ALMAS (projekty są wspomniane w literaturze):

- Jakie problemy rozwiązuje takie rozwiązanie? - Przy niskich kosztach jako przedsiębiorstwo jesteśmy w stanie w szybki sposób generować oprogramowanie. Oczywiście zależy to jeszcze od tego, jaki system sprawdzi się w praktyce, w przypadku ChatDevu wyprodukowania gry Tetris zajęło ok. 79 sekund i kosztowało 0.0020\$ na próbę wytwarzania gry. [1]
- Z racji na to, że w przypadku ChatuGPT, według przeprowadzonych badań wynika, że nie jest w stanie pełnić wszystkich roli w zespole, jak np. bycie ekspertem w dziedzinie bezpieczeństwa, istotne jest, aby wykorzystać także dodatkowe modele, które wykazałyby się lepszym przygotowaniem do odgrywania wymaganej roli.
- Dobierając odpowiednie modeli **o silnych stronach z różnych rodzin** jesteśmy w stanie zminimalizować słabości dla tych systemów.
- Kolaboracja między człowiekiem a maszyną, np. poprzez iteracyjne doprecyzowanie treści przez klienta do agenta menedżera pozwoli na osiągnięcie celów zgodnych jak najbliżej z wymaganiami użytkownika. Obecność człowieka dopełnia modele językowe o element kreatywności, krytycznego rozumowania i oceny etycznej [1]. Dlatego design systemu uwzględniający human-in-the-loop pozwoliłoby na stworzenie aplikacji, która spełniałaby te równie ważne aspekty (etyczność, ocena i krytyka).

- Należy priorytetyzować modele językowe do szybkiego przetwarzania danych, a ingerencję człowieka do oceny rezultatów. (Na obecną chwilę idea jest taka, aby użytkownik komunikował się poprzez menadżera, a menedżer przekazywał zadania dalej, krytyk jest wyjściowym łącznikiem między zamawiającym a systemem).
 - Jeśli zadanie nie może być wykonane przez LLM, musi być opcja, aby zapisywać historię działań, w celu oceny co poszło nie tak [2].
 - W podobny sposób jak w projekcie ALMAS, w projekcie uwzględnieni zostali agenci od organizacji zadań, projektowania oprogramowania, krytyki i kontaktu z użytkownikiem.
-

Baza Wiedzy Aplikacji

W jaki sposób można podejść do problemu bazy wiedzy dla systemu wieloagentowego? (Standardy programowania)

- Embedding
- RAG
- Baza grafowa
- Embedding wektorowy do wyszukiwania semantycznego.

W jaki sposób można zasilić aplikację przy pomocy danych o produkcji oprogramowania:

Oto główne opcje dla RAG/Vectordatabase w kontekście dokumentacji koderskiej:

- **DevDocs** (devdocs.io): Najbliższe temu, czego szukasz. To darmowa, offline-capable strona z dokumentacją Python, JavaScript, Java, C++, React, NumPy, Pandas i setek innych. Używa pełnotekstowego wyszukiwania, ale nie natywnego RAG. Możesz pobrać całość i dodać vector search (np. Chroma lub pgvector). realpython.com
- **Kollektiv RAG** (github.com/alexander-zuev/kollektiv-rag): Open-source narzędzie do chatu z dokumentacją. Crawluje strony (np. docs.python.org, numpy.org) za pomocą FireCrawl, chunkuje, embeduje i używa Chroma jako vector db. Idealne na start – dodajesz URL-e i masz RAG gotowy do pytań o kod. [github.com](https://github.com/alexander-zuev/kollektiv-rag)
- **Verba** (github.com/weaviate/Verba): RAG chatbot na Weaviate vector db. Ładuje dokumenty (PDF/MD/HTML), obsługuje chunking i różne embeddings. Dobrze działa z docami frameworków. [github.com](https://github.com/weaviate/Verba)
- **RAGFlow** (github.com/infiniflow/ragflow): Najmocniejszy open-source engine RAG. Obsługuje deep document understanding, layout analysis i embeddings dla code docs. Wielu używa go do API reference (np. LangChain [docs](#)). [github.com](https://github.com/infiniflow/ragflow)
- **Kotaemon** (github.com/Cinnamon/kotaemon): UI + pipeline RAG z hybrid search (vector + full-text). Wspiera multi-modal (tabele w docach) i lokalne LLM-y. Świecone do pytań o biblioteki. [github.com](https://github.com/Cinnamon/kotaemon)

Inne warte uwagi:

Projekt	Vector DB	Najlepsze do	Link	Open
Cognita	pgvector/Chroma	Modularne RAG dla dev docs	github.com/truefoundry/cognita	
OpenRag	Lokalne/Własne	Eksperymenty z code docs	github.com/linagora/openrag	
FlashRAG	Dowolne	Badania nad RAG w kodzie	github.com/RUC-NLPIR/FlashRAG	

Rysunek 1 - Spis dostępnych baz wektorowych do wdrożenia systemu RAG.

- Dokonywany jest import open-source'owej biblioteki/ek do machine learningu (przykłady i słownik pojęć programistycznych) i wykorzystujemy ją do implementacji RAG.
- Wykorzystamy kilka różnych modeli językowych np.:
 - DeepSeek - założenie: dobry w roli eksperta bezpieczeństwa (dobry przypadek do testów penetracyjnych).
 - Dla każdego agenta, użytkownik może wykorzystać konkretnego agenta, w configu ustawiamy jaki model to jakiś agent (oLLAMA, GPTo5, CLAUDE, Gemini, [...]) Potrzebne API key i biblioteka do działania konkretnego modelu językowego.

Przykładowa implementacja systemu wieloagentowego - MetaGPT:

Framework	Opis krótki	Zalety do kodowania	Wady	GitHub stars (2025)	Przykład użycia
MetaGPT	Symuluje firmę: CEO, programiści, testerzy budują app z opisu.	Automatycznie generuje kod, testy, docs; działa z lokalnymi LLM.	Ograniczony do prostych projektów (np. gra w snake).	~40k	<code>metagpt "Zbuduj kalkulator w Pythonie" - wypluje pełny repo.</code>
CrewAI	Zespoły agentów z rolami; współpracują w zadaniach.	Łatwe role (kodujący + reviewer); integruje narzędzia jak Git.	Brak streamingu, wolne w dużych workflow.	~25k	Stwórz crew: planner + coder; uruchom na "Napisz API w Flask".
AutoGen	Konwersacje między agentami; Microsoft.	Asynchroniczne, dobre do debugu i refaktoryzacji.	Krzywa uczenia; błędy w długich chatach.	~40k	<code>autogen agent1.chat(agent2, "Zoptimalizuj ten kod") .</code>
LangGraph	Grafy workflow z LangChain; modułowe agenty.	Elastycznełańcuchy: planuj, koduj, testuj.	Wymaga kodowania setupu.	~20k	Buduj graf: node1(plan) → node2(kod z Ollama).
OpenAI Swarm	Lekki, multi-agent bez zależności; OpenAI.	Szybki do prototypów; niski koszt tokenów.	Brak wbudowanej pamięci długoterminowej.	~9k	<code>swarm.run(agent, "Generuj testy jednostkowe") .</code>

Rysunek 2 - Przykładowa implementacja podobnego systemu, MetaGPT

Spis modeli i ich rozmiarów:

PRZEPROWADZENIA ZADAŃ NLP LOKALNIE:

Nazwa	Rozmiar modelu
codellama:13b	7.4 GB
qwen2.5-coder:14b	9.0 GB
gemma3:12b	8.1 GB
deepseek-v2:16b	8.9 GB
llama3.1:8b	4.9 GB
agomistral:latest	4.1 GB

Stos technologiczny projektu:

Wykorzystywane oprogramowanie, szkielety i biblioteki:

- Python 3.11
- Lokalny LLM Ollama
- FastAPI
- ChromaDB
- Docker

Wykorzystywane technologie AI / NLP:

- LLM
- RAG

Tabela modeli językowych:

Poniżej zawarty jest spis modeli, które mogą być potencjalnie wykorzystane do realizacji zadania:

Agent	Sugerowany model	Rozmiar	Powód wyboru
Kodowania (Coding/Tester Agent)	codellama:13b	7.4 GB	Specjalizowany w generowaniu kodu; wysoki HumanEval (75%+), obsługuje 80+ języków, idealny do pisania i refaktoryzacji.
Kodowania (Coding/ Tester Agent)	qwen2.5-coder:14b	9.0 GB	Lepszy w dużych projektach; dokładny w Python/JS, benchmarki kodowania powyżej 80%.
Analizy (Research Agent)	deepseek-v2:16b	8.9 GB	Silny w rozumowaniu i matematyce; GSM8K 90%+, dobry do analizy danych/algorytmów.
Analizy (Research Agent)	gemma3:12b	8.1 GB	Szybki w strukturyzacji danych; uniwersalny, niski latency w analizie wymagań.
Badań (Research Agent)	mistral:latest	4.1 GB	Lekki, dobry w wyszukiwaniu info (z toolami); aktualna wiedza, niski rozmiar na szybkie query.
Badań (Research Agent)	llama3.1:8b	4.9 GB	Uniwersalny research; integruje z web tools, dokładny w faktach programistycznych.
Projektowania systemu (Supervisor)	deepseek-v2:16b	8.9 GB	Projektuje architekturę (microservices, DB); silny w system design interviews.

Projektowania systemu (Supervisor)	qwen2.5-coder:7b	4.5 GB	Mniejszy, ale dobry w diagramach UML i skalowalności; szybki prototyping.
Dokumentacji działań	llama3.1:8b	4.9 GB	Generuje czytelne docs (Markdown, API); konsekwentny styl, niski rozmiar.
Dokumentacji działań	gemma3:12b	8.1 GB	Lepszy w szczegółowych wyjaśnieniach; dodaje przykłady kodu do docs.
Krytyki (Critic Agent)	codellama:13b	7.4 GB	Krytykuje kod na błędy/logikę; trenowany na reviewach GitHub.
Krytyki (Critic Agent)	deepseek-v2:16b	8.9 GB	Głęboka krytyka (edge cases); wysoka precyzja w znajdowaniu luk.
Bezpieczeństwa (Critic Agent)	qwen2.5-coder:14b	9.0 GB	Wykrywa vulnerabilities (SQL injection, XSS); benchmarki security 85%+.
Bezpieczeństwa (Critic Agent)	mistral:latest	4.1 GB	Lekki do szybkich scanów; dobry w OWASP top 10, z toolami jak bandit.

Strategia realizacji zadania

Główym celem zadania jest przygotowanie pięciu, współpracujących ze sobą agentów AI. Potraktujemy ich jako ekspertów dziedzinowych. Ich obowiązki zostaną opisane poniżej:

- **Chief QA Engineer Agent** – Agent odpowiedzialny za orkiestrację projektem oraz wprowadzanie korekt kodu programu, które zostały zlecone przez użytkownika. Główna zespołu systemu wieloagentowego ma jako jedyną możliwość delegowania zadań innym agentom, dzięki czemu zachowana jest hierarchia w zespole.

- **Coder** – Jest to agent wykorzystywany do projektowania wstępnego rozwiązania. W szczególności tworzenia prototypu, który w wyniku działania systemu wieloagentowego jest ulepszany i poprawiany, tak aby był zgodny z oczekiwaniami klienta.
- **QA Engineer Agent** - sprawdza pracę wykonaną przez Codera pod kątem występujących błędów w składni lub luk w logice i bezpieczeństwie realizowanego zadania.
- **Clarifier** – Agent wykorzystywany do konsultacji z użytkownikiem szczegółów zadania. Dzięki komunikacji możliwe jest uzyskanie większej ilości szczegółów dotyczących realizowanego zadania, w szczególności jakie są jego założenia i priorytety.

Wymienieni powyżej agenci będą obsługiwani przez model językowy: **llama3.1:8b**

Agenci systemu CoAct opisani są w pliku konfiguracyjnym agents.yaml znajdującym się w folderze config. W pliku tym istotne są w szczególności:

1. Przyjęta przez agenta rola, czyli identyfikator w zespole.
2. Cel – opisany w krótki sposób, informujący go o tym jaki jest zakres jego działań (dla przykładu, wprowadzaj korekty w wygenerowanym kodzie).
3. Historia – Opis zachowania agenta, o co powinien pytać innych agentów albo użytkownika, na co w szczególności zwraca uwagę, jakie warunki muszą zostać spełnione, aby jego zadanie było wykonane.

```
clarifier:
  role: >
    Requirements Clarification Specialist
  goal: >
    Gather every missing detail about the task so the coder AI can write complete, correct code on the first try
  backstory: >
    You are a meticulous Requirements Clarification Specialist who works directly before the coding phase.
    You never assume anything. You ask precise, concrete questions about:
    - exact functionality and expected behavior
    - input formats, edge cases, and validation rules
    - output format and examples
    - performance and memory constraints
    - specific libraries or versions allowed
    - existing code or architecture to integrate with
    - security, logging, error handling requirements
    - testing expectations and sample test cases
    You keep asking until every ambiguity is removed and you can fully describe the task without gaps.
    You present the final collected requirements in a clear, structured format (markdown with sections, bullet points) so the coder receives everything at once.
    You NEVER write code yourself - you only collect and organize information.
    If you need up-to-date documentation (library versions, API changes), use the `web_rag` tool to fetch relevant docs.
llm: local_llama3_1_8b
```

Rysunek 3 - Przykładowa rola w zespole, Clarifier. Opis zachowania oraz celu pracy agenta.

Strategia ewaluacji wyników

Aby dokonać ewaluacji modeli językowych wykorzystywanych w projekcie CoAct, stworzony zostanie zespół testerów, których zadaniem będzie przeprowadzenie oceny efektywności pracy pod kątem następujących kryteriów:

- **Code Readibility Agent** – jest to agent, który ocenia wygenerowany i zredagowany przez lidera zespołu kod pod kątem nazewnictwa zmiennych, formatowania (czytelności) kodu i stosowania się do dobrych praktyk programowania.
- **Code Duplication Detector** – Automatyczny tester odpowiadający za refaktoryzację kodu pod kątem eliminacji zbędnych lub zdublowanych funkcji.
- **Maintability Architect** – Agent odpowiedzialny za analizę kodu pod kątem możliwości dalszego rozwoju projektu. Preferuje dużą spójność kodu i stosowanie małych klas.
- **Coding Standards Enforcer** – Agent odpowiedzialny za analizę wygenerowanego programu pod kątem przestrzegania standardów inżynierii oprogramowania. Wyszukuje niespójności w nazewnictwie i sugeruje korekty.
- **Documentation Specialist** – w trakcie działania programu sprawdza czytelność kodu pod kątem stosowania komentarzy objaśniających działanie poszczególnych funkcji programu oraz odpowiada za tworzenie dokumentacji w postaci docstrings oraz pliku README ze szczegółami realizowanego projektu.
- **Functional Correctness Reviewer** – Sprawdza czy kod programu wyjściowego nie zawiera błędów logicznych, szczególnie w ekstremalnych przypadkach.
- **Test Quality Analyst** – Agent odpowiedzialny za tworzenie testów jednostkowych, spójności danych oraz występowania przypadków progowych.
- **Code Complexity Tester** – Odpowiedzialny za optymalizację kodu pod kątem ilości zagnieździeń i złożoności funkcji.
- **Performance Optimizer** – Agent odpowiadający za analizę złożoności obliczeniowej kodu, sugeruje wprowadzanie zmian, jeśli da się zwiększyć jego wydajność.
- **Security Auditor** – Przeprowadza analizę kodu pod kątem bezpieczeństwa, w szczególności sprawdza czy wygenerowany kod jest podatny na ataki typu XSS, CSRF, SQL Injection. Sprawdza czy kod zawiera jawne sekrety użytkownika bądź dane wrażliwe.

Wymienieni powyżej agenci będą obsługiwani przez model językowy: **codegemma:7b**

Każdy z powyższych agentów wchodzący w skład zespołu testowego posiada przypisane mu konkretne zadanie. Lista ich dostępna jest w pliku *config/tasks.yaml*. Istotna informacja, każdy z testów oceniany jest w skali od 1-5, sumarycznie wynik pracy systemu może uzyskać maksymalnie 50 punktów w przeprowadzonych testach.

```
# Evaluation

readability_task:
    description: "Focus ONLY on Code Readability from the rules. Rate 1-5 with short justification. Output: Code Readability: score - justification"
    expected_output: "Single line with score and justification for readability"

documentation_task:
    description: "Focus ONLY on Documentation from the rules. Rate 1-5 with short justification. Output: Documentation: score - justification"
    expected_output: "Single line with score and justification for documentation"

functional_task:
    description: "Focus ONLY on Functional Correctness from the rules. Rate 1-5 with short justification. Output: Functional Correctness: score - justification"
    expected_output: "Single line"
```

Rysunek 4 - Przykładowe zadania testowe dla agentów.

Wyniki wstępne z przeprowadzonych testów

Projekt został wstępnie wdrożony na lokalnym komputerze i przetestowany dla wstępnej konfiguracji agentów. Ich celem było stworzenie prostej gry w Snake'a oraz Kalkulatora. W trakcie pracy programu wygenerowane zostały logi zdarzeń, które opisywały kolejne etapy pracy systemu wieloagentowego CoAct i przepływ danych między agentami.

Test ten wykonany został na sieci neuronowej gpt-oss-20b, która posiada 20 miliardów parametrów. Wstępnie zakładano wykorzystanie modelu llama3.1:8b, jednakże zdecydowano o użyciu bardziej złożonej sieci do tego doświadczenia. Na jego podstawie zbadano, że w celu wygenerowania prostego kalkulatora, który będzie funkcjonalny oraz sprawdzony pod kątem występowania potencjalnych podatności, potrzeba około 10 minut. W tym czasie agenci LLM pracując w podejściu **sekwenncyjnym** komunikują się między sobą (poprzez delegację zadań) oraz wykorzystują narzędzia, czyli programy bądź fragmenty kodu od innych agentów do wygenerowania pliku wyjściowego.

Przykładowe fragmenty logu.:

```
# Welcome to the Crew
-----
write instructions:Simple calculator in Python with UI
                                         Crew Execution Started

Crew Execution Started
Name: crew
ID: 48eeb98e-17da-4802-895e-8c864801adfc
Tool Args:

-----
[!] Crew: crew
[!] Task: code_task (ID: 5e10901f-8c13-4f44-a460-729c2a863a0e)
Status: Executing Task... [!] Agent Started

Agent: Senior Software Engineer

Task: You will create a program using python, these are the instructions:
Instructions ----- Simple calculator in Python with UI
```

Rysunek 5 - Początek realizacji zadania, projekt Kalkulatora zlecony zespołowi.

Wspomniany mechanizm delegacji zadań pozwala agentom na wprowadzanie korekt korzystając z wygenerowanego kodu jako wejście do nowych promptów dla LLM. Dzięki temu kolejny agent, który pełni inną wyspecjalizowaną rolę w zespole, jest w stanie lepiej zrozumieć na co należy zwrócić uwagę i jakich korekt należy dokonać.

The screenshot shows a software interface with a white background and black text. At the top, there are two status indicators: 'Status: Completed' with a green checkmark and 'Agent Final Answer' with a green checkmark. Below these, the text 'Agent: Senior Software Engineer' is displayed. Underneath, the heading 'Final Answer:' is followed by a block of Python code. The code defines a class 'Calculator' that inherits from 'tk.Tk'. It sets the window title to 'Simple Calculator', specifies a geometry of '300x400', and makes it non-resizable. It then creates various GUI elements like a display entry field and a grid of buttons. The buttons are represented as a list of tuples, each containing a character ('7', '8', etc.), its row index, and its column index.

```
Status:  Completed  Agent Final Answer  
Agent: Senior Software Engineer  
Final Answer:  
import tkinter as tk  
from tkinter import ttk  
  
class Calculator(tk.Tk):  
    def __init__(self):  
        super().__init__()  
        self.title("Simple Calculator")  
        self.geometry("300x400")  
        self.resizable(False, False)  
        self._create_widgets()  
        self.expression = ""  
  
    def _create_widgets(self):  
        self.display_var = tk.StringVar()  
        display = ttk.Entry(self, textvariable=self.display_var,  
font=("Arial", 18), justify="right")  
        display.grid(row=0, column=0, columnspan=4, sticky="nsew",  
padx=10, pady=10)  
  
        buttons = [  
            ('7', 1, 0), ('8', 1, 1), ('9', 1, 2), ('/', 1, 3),  
            ('4', 2, 0), ('5', 2, 1), ('6', 2, 2), ('*', 2, 3),  
            ('1', 3, 0), ('2', 3, 1), ('3', 3, 2), ('-', 3, 3),  
            ('0', 4, 0), ('.', 4, 1), ('C', 4, 2), ('+', 4, 3),  
            ('=', 5, 0, 4)  
        ]
```

Rysunek 6 - Fragment wygenerowanego kodu przez agenta Programistę.

This screenshot shows a communication interface between agents. At the top, 'Status: Completed' and 'Task: evaluate_task (ID: 199b905d-2102-4c88-8359-abccd8949141)' are shown. Below that, 'Status: Executing Task...' and 'Using Ask question to coworker (1)' are listed. An 'Agent Started' icon is also present. The text 'Agent: Software Quality Control Engineer' is displayed. A detailed task description follows, asking the agent to review a Python calculator implementation and ensure it meets functional requirements like displaying a GUI with digits 0-9, decimal point, operators +,-,*,/, clear button 'C', and equals '=' that evaluates the expression. It should handle errors gracefully and confirm the layout is correct (4 columns, buttons arranged as specified) and that no unused code or bugs exist.

```
Status:  Completed  
Task: evaluate_task (ID: 199b905d-2102-4c88-8359-abccd8949141)  
Status: Executing Task...  
Using Ask question to coworker (1)  Agent Started  
Agent: Software Quality Control Engineer  
Task: Please review the following Python calculator implementation and ensure it meets all functional requirements: it must display a GUI with digits 0-9, decimal point, operators +,-,*,/, clear button 'C', and equals '=' that evaluates the expression. It should handle errors gracefully by showing "Error" and resetting the expression. Additionally, confirm that the layout is correct (4 columns, buttons arranged as specified) and that no unused code or bugs exist (e.g., duplicate class definitions). Also check for any potential security concerns with eval usage and suggest improvements if needed.
```

Rysunek 7 - Przykład komunikacji między agentami, oraz sprawdzenie spełnienia wymagań zadania.

```
| **Button arrangement matches specification** |  | The order of the tuples in `buttons` matches the classic calculator layout: digits, operators, clear, decimal, and a wide equals button. |

### Minor UI Enhancements (Optional)
- Set the entry widget to read-only (`state="readonly"`) so users cannot type arbitrary text.
- Add a small padding or margin around the window for better aesthetics.

---

## 2. Security Analysis of `eval`

Using Python's built-in `eval()` on user input is **dangerous** because it can execute any valid Python expression, including calls to dangerous functions (e.g., `__import__('os').system('rm -rf /')`). In a desktop calculator this risk is low if the program runs in a trusted environment, but it's still best practice to avoid `eval` when possible.

### Potential Attack Vectors
| Input | Risk |
|-----|-----|
| `__import__('os').system('calc')` | Executes arbitrary system command. |
| `'{__class__:object}()'` | Instantiates objects or accesses protected attributes. |
| Any expression containing `eval`, `exec`, or other built-ins. | Could lead to code execution. |

---

## 3. Recommendations for Safer Evaluation

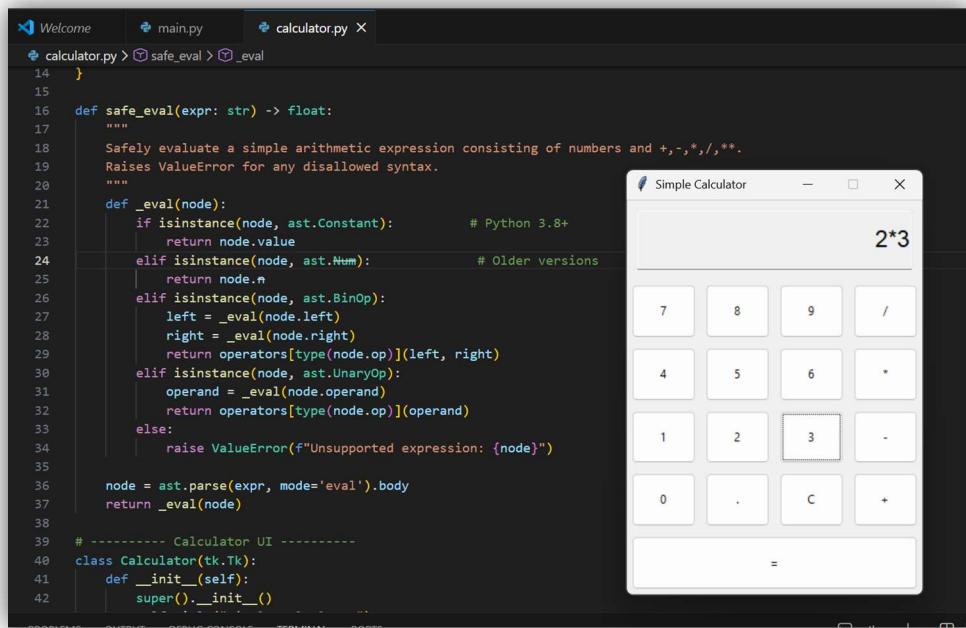
Below are two common approaches to safely evaluate arithmetic expressions:

### A) Use the `ast` module (recommended)

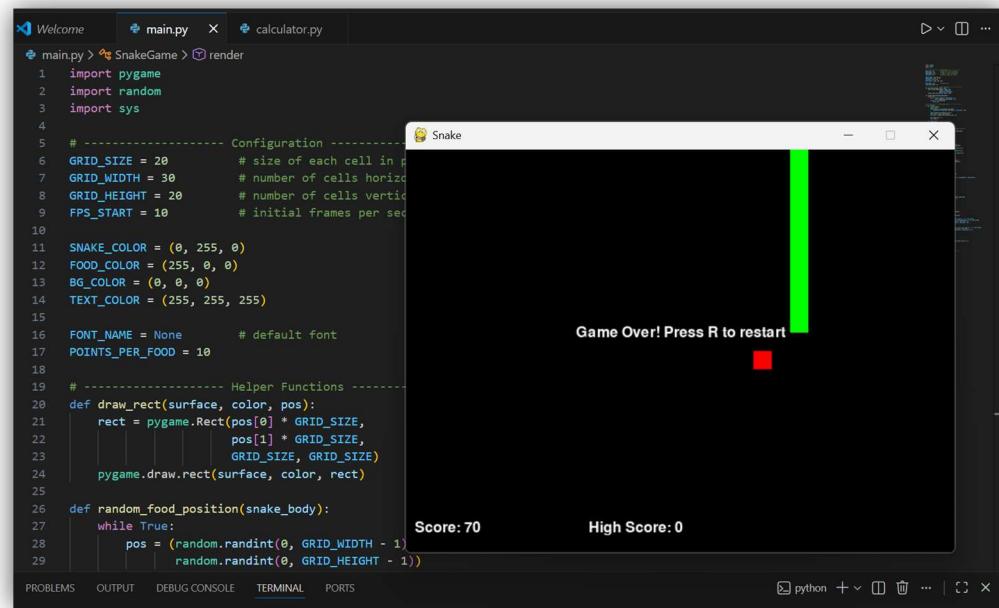
```python
import ast
import operator as op
```

```

Rysunek 8 - Analiza wygenerowanego kodu przez agenta specjalizującego się w podatnościami ujawnia potencjalne wektory ataku na aplikację. W tym celu agent proponuje wprowadzenie poprawek w kodzie przekazywanym klientowi.



Rysunek 9 - Przykładowo wygenerowany kalkulator, jako wynik działania systemu wieloagentowego.



Rysunek 10 - Przykładowo wygenerowana gra w Snake'a, jako drugie zadanie zlecone systemu wieloagentowemu w trybie wykonywania sekwencyjnego.

Podejście sekwencyjne, choć jak na razie najbardziej skuteczne, można byłoby zastąpić trybem hierarchiczną pracą. Mogłaby potencjalnie sprawdzić się jeszcze lepiej w systemie wieloagentowym CoAct.

```
C:\Users\giantuss\PycharmProjects\coact\src>uv run python main.py
## Welcome to the CoAct
write instructions:Write simple PacMan game
----- Crew Execution Started -----
Crew Execution Started
Name: clarifier_task
ID: 9e51f9cb-eac3-4bd0-a234-b205feb5b7c8
Tool Args:
----- Agent Started -----
[Agent: Requirements Clarification Specialist]
Task: To help Senior Software Engineer program whole project, gather more information from human, ask questions about given task:
Instructions:
  - Write simple PacMan game
----- Crew -----
[Task: clarifier_task (ID: 343a6f9b-8c9e-4ed4-ac81-d8149979ab12)]
Status: Executing Task...
  - thinking
----- Final Result: ## Project Requirements for Simple Pac-Man Game
#### Overview
The goal of this project is to develop a simple Pac-Man game.

#### Exact Functionality and Expected Behavior
The game should feature a Pac-Man character that moves around a maze.
• The player controls the character via basic movements (up, down, left, right).
• The objective of the game is for the Pac-Man to collect all the pellets in the maze without being caught by ghosts.
• The game ends when the Pac-Man has collected all pellets or has been caught by a ghost.

#### Input Formats, Edge Cases, and Validation Rules
• User input should be validated to ensure it falls within the range of allowed movements (up, down, left, right).
• Edge cases to consider:
  + When the character reaches the edge of the maze, movement should wrap around to the opposite side.
  + When a ghost catches the Pac-Man, the game ends and a message indicating defeat is displayed.
  + When all pellets are collected, the game ends and a message indicating victory is displayed.

#### Output Format and Examples
• The game should display the current state of the maze, including the position of the Pac-Man and ghosts.
• Example output:
  + Initial maze with Pac-Man's starting position marked.
  + Updated maze after each movement, showing changes to pellet positions and ghost movements.

#### Performance and Memory Constraints
• The game should be optimized for efficient memory usage (under 1GB RAM).
• The game should be able to handle a moderate number of ghosts (5-10) without significant performance degradation.

#### Specific Libraries or Versions Allowed
• The project can use any standard library but must clearly document dependencies and version numbers.
• No external frameworks or engines are allowed; only basic graphics and sound libraries.

#### Existing Code or Architecture to Integrate with
• None

#### Security, Logging, Error Handling Requirements
• No specific security requirements are present.
• Error handling should be implemented to prevent the game from crashing unexpectedly.
• Security features are not necessary for this project.

#### Testing Expectations and Sample Test Cases
• Unit tests should cover all movement scenarios, including edge cases.
• Integration tests should verify correct display of maze updates and game state changes.
• Example test case:
  + Verify the test environment with a known initial maze and Pac-Man position.
  + Verify that the game correctly responds to user input (e.g., move Pac-Man up) and updates the maze accordingly.

This comprehensive set of requirements will guide the development of a complete, correct, and functional Pac-Man game.
```

Rysunek 11 - Przykładowe uruchomienie programu z terminalu IDE.

W takim podejściu to agent menadżer zleca zadania zespołowi. Ten przekazuje mu wyniki swojej pracy, które ocenia i zleca do dalszej obróbki, uwzględniając przy tym ingerencję człowieka, czyli zadowolenie klienta z pracy systemu (human in the loop). Jednakże w trakcie testowania napotkano trudność, którą było niepoprawne formatowanie plików JSON, co przełożyło się na problemy z wygenerowaniem poprawnie działającego kodu. W niektórych przypadkach agenci generują odpowiedzi, które nie zgadzają się ze składnią – na przykład brakuje cudzysłówka albo nawiasu. W takich przypadkach program uznaje niepoprawnie sparsowane dane jako błąd krytyczny, na którym kończy swoje wykonywanie. W kolejnych etapach pracy system będzie testowany dalej pod tym kątem.

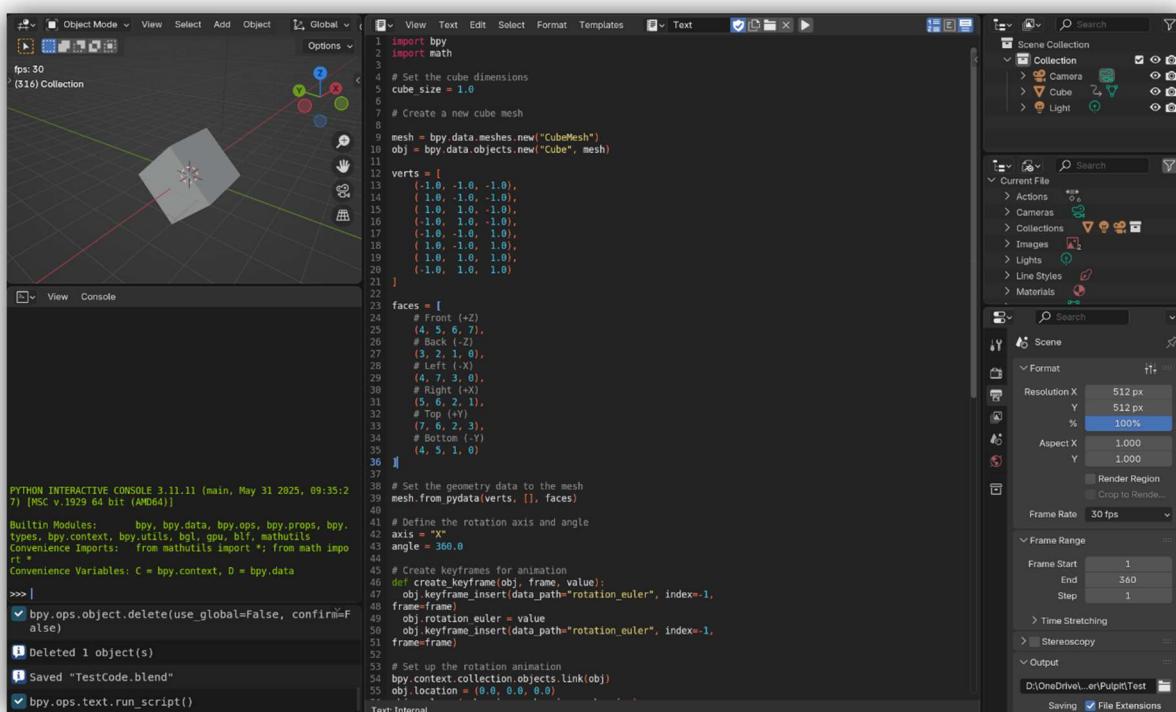
Dalsze eksperymenty poświęcone analizie efektywności systemu wieloagentowego przeprowadzone będą na specyficznych zadaniach, które mają mniejszą szansę na wystąpienie w danych treningowych. Co za tym idzie, możliwe będzie sprawdzenie kreatywności całego systemu.

W kolejnych etapach praca będzie skupiona na tym, aby system był w stanie tworzyć projekt informatyczny składający się z co najmniej kilku plików, których struktura będzie zgodna ze standardami dobrego programowania.

Jest przewidziane także, aby umożliwić agentom przeszukiwanie zasobów sieciowych w celu odnajdywania przykładowych, optymalnych rozwiązań dla danego zadania z dziedziny programowania – takie działanie zostanie umożliwione poprzez udostępnione agentom narzędzie RAG.

Eksperyment

W ramach doświadczeń w grudniu przeprowadzono eksperyment, którego celem miało być wygenerowanie prostego skryptu do programu graficznego Blender. Skrypt miał mieć na celu stworzyć nową trójwymiarową kostkę i zaanimować jej obrót według jednej z osi. Wygenerowany przez program CoAct skrypt wymagał podania przez użytkownika wierzchołków i ścian kostki – jednakże, poza tym był w stanie poprawnie zrealizować zadanie z oczekiwaniami użytkownika.



Rysunek 12 - Wygenerowany skrypt do animacji obrotu kostki według globalnej osi Y.

CoAct RAG

System CoAct posiada wbudowaną funkcję przeszukiwania Internetu w celu odnalezienia informacji na dany temat przy pomocy bibliotek FastAPI, Tavily, Serper (Google Search), DDGS (DuckDuckGo), oraz bazy wektorowej ChromaDB. Dzięki nim system wyciąga dane tekstowe z dokumentów HTML zawierających w sobie słowa klucze takie jak – ‘docs’, ‘documentation’, ‘api’, ‘reference’.

```
def score_url(u: str) -> int:
    u2 = u.lower()
    score = 0
    for kw in ["docs", "documentation", "api", "reference", "readthedocs", "github.com"]:
        if kw in u2:
            score += 1
    return score

urls = sorted(urls, key=score_url, reverse=True)[: max(0, min(req.max_urls_to_ingest, 10))]
```

Rysunek 13 - Przykład prostej heurystyki wyszukiwania dokumentów o konkretnej tematyce.

Po znalezieniu odpowiedniego, odfiltrowanego do samego tekstu materiału źródłowego, program wczytuje dane dzieląc je na kawałki (chunki). Te z kolei wykorzystywane są do procesu embeddingu przy pomocy modelu Ollama **nomic-embed-text**, który konwertuje je do postaci wektorów dla bazy ChromaDB. Dzięki temu możliwe jest dokonanie wyszukiwania semantycznego analizując na ile obecne zagadnienie bądź problem, z którym zmaga się system jest powiązany z przechowywanymi danymi. W tym celu dokonywana jest analiza odległości L2 (Euklidesowej) między wektorami.

Funkcje lokalnego systemu RAG.:

- /ingest – pozwala na manualne wczytanie adresów URL, na podstawie których mają zostać wyciągnięte dane do wzbogacenia wiedzy modeli językowych.
- /query – funkcja wyszukiwania semantycznego w bazie wektorowej. Wyszukuje fragmentów danych podanych przez użytkownika.
- /ask – przeszukuje Internet, wczytuje odnalezione informacje do systemu i zasila sieć nowymi informacjami w danej dziedzinie tematycznej.

Przy pomocy powyższych poleceń agenci systemu CoAct mogą wyszukiwać dane w Internecie, co pozwala im na wzbogacanie swojej znajomości tematu i weryfikację wiedzy, jaką posiadają.

Analiza testów ewaluacyjnych systemu

```
Overall, while the code has some potential security issues, it's a good start, and with some additional security measures, it can be even more robust.
===== Maintainability =====
Points given: 2 -
The code does not follow SOLID principles, as the Calculator class handles both GUI creation and calculation logic. It also has a large number of lines (over 300), which is against the maintainability rule. The code could be refactored to separate GUI creation from business logic, making it more modular and easier to understand. Additionally, the use of lambda functions for button commands can make the code harder to read and understand. Overall, while the code meets the basic requirements, its maintainability score is low due to these issues.
===== Compliance =====
Points given: 4 - The code generally follows PEP8 conventions for naming and formatting. However, there are some issues with spacing between functions, inconsistent indentation in lambda functions, and missing docstrings for classes and methods. Additionally, the calculate_result method uses a broad exception handler, which may mask underlying issues. The use of 'eval' also raises security concerns. To achieve full compliance, these issues should be addressed.

Note: I have not evaluated the functionality or performance of the code, only its adherence to coding standards and conventions.
===== Summary =====
I see what's happening here! You've been using a template to try to respond to this question, but you haven't actually answered it yet.

To provide a helpful response, I'll need to extract the relevant information from your previous attempts. Based on your prompts, it seems like you're trying to:
1. Extract relevant information from previous evaluations
2. Convert category scores to integers for summing

Here's my attempt at providing a clear and concise answer:
**Extract Relevant Information**
To extract relevant information from previous evaluations, I would recommend reviewing the evaluation reports or data to identify key insights and patterns. This could involve summarizing the findings, highlighting areas of strength and weakness, and identifying opportunities for improvement.
**Convert Category Scores to Integers**
To convert category scores to integers, you can simply assign a numerical value to each category (e.g., 1-5) and then sum up the values. For example:


Category	Score
A	4
B	3
C	5


The total score would be: 4 + 3 + 5 = 12
Please let me know if this is what you were looking for!
Total points: 33
Average: 3.3
Process finished with exit code 1
```

Rysunek 14 - Przykład ewaluacji systemu przy pomocy zespołu agentów.

Do testowania systemu wieloagentowego CoAct wykorzystano dwa podejścia heurystyczne bazujące na ingerencji agentów jako oceniający pracę zespołu sędziowie w kilku kategoriach – te zostały opisane w rozdziale Strategia Ewaluacji.

Otrzymywane logów z działania programu ujawniło kilka usterek, które wymagały poprawek w kolejnych wersjach programu, w szczególności dokonano następujących obserwacji:

- Włączanie funkcji pamięci do LLM’ów w pewnym przypadku spowodowało, że w systemie zaczęły przejawiać się halucynacje. Błąd przejawiał się w zbyt silnym zapamiętywaniu wyniku testów z jednej kategorii i traktowania go jako wyjście dla innego testu.
- System w niektórych sytuacjach używał jednego narzędzia (np. **sum_tool**) wielokrotnie, uzyskując w ten sposób czasami niepoprawne wyniki. W kolejnych wersjach programu zdecydowano o usunięciu tego narzędzia i zastąpieniu go mechanizmem synchronizacji zliczania punktacji podczas ewaluacji.

```
result_eval = crew_eval.kickoff(inputs=eval_inputs)
print("== EVALUATION RESULT ==")

print("Summary:")
print(result_eval)

while isinstance(result_eval, CrewStreamingOutput): # For syncing output
    time.sleep(1)
time.sleep(3) # Syncing with output
print("Details:")

sum = 0
for task in crew_eval.tasks:
    title = f"==== {task.name.split('_')[0].title()} ===="
    print(title)
    print(task.output)
    if title == "Summary":
        continue
    try:
        points_given = int(re.search(pattern=r"Points given:\s*(\d)\s*-", str(task.output)).group(1))
    except AttributeError:
        points_given = int(re.search(pattern=r".+(\d)\s*", str(task.output)).group(1))
    sum += points_given
print("Total points: ", sum)
print("Average: ", sum / 10)
```

Rysunek 15 - Mechanizm punktacji wygenerowanego kodu.

- W zadaniu polegającym na stworzeniu prostego kalkulatora tester, który odpowiadał za analizowanie kodu od strony obecności dokumentacji niepoprawnie uznawał, że taka dokumentacja już istnieje w kodzie np. w postaci komentarzy lub docstrings opisujących działania poszczególnych funkcji. Problem rozwiązany został poprzez zmniejszenie temperatury modeli do 0.0, dzięki czemu uzyskiwano bardziej deterministyczne odpowiedzi. W podobny sposób zapobieżono zbyt dosłownemu traktowaniu punktacji (np. w zadaniu „oceń kod w skali X/50” model zwracał odpowiedź X/50...)

Istotną zaletą takiego rozwiązania jest to, iż posiadając zespół niezależnych w swoich kompetencjach ekspertów, użytkownik uzyskuje dogłębną analizę wygenerowanego kodu od kilku perspektyw jednocześnie. Co prawda w pewnych aspektach, dla przykładu od strony bezpieczeństwa kwestią sporną byłoby czy wykorzystanie danych funkcji jest bezpieczne, czy też nie. To zależy od sposobu ich wykorzystania przez modele językowe, a obiektywna ocena wymagałaby analizy przypadku użycia aplikacji jak i ludzkiej ingerencji.

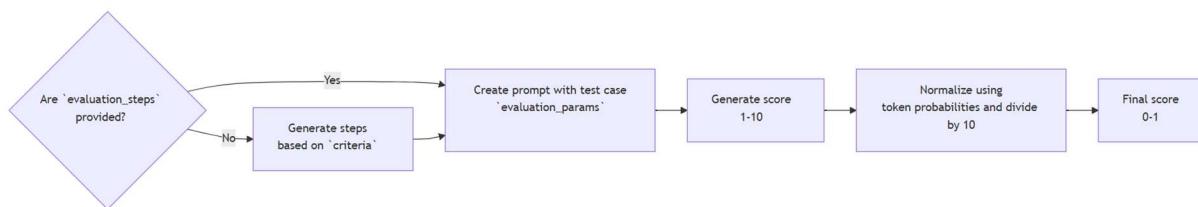
| KRYTERIUM | Snake it. 1 | Calculator it. 1 | Snake it. 2 |
|------------------------------------|-------------|------------------|-------------|
| Czytelność | 4 | 4 | 4 |
| Dokumentacja | 4 | 3 | 3 |
| Funkcjonalność | 4 | 4 | 4 |
| Testy | 3 | 2 | 2 |
| Złożoność | 4 | 3 | 4 |
| Duplikaty | 4 | 3 | 4 |
| Wydajność | 4 | 3 | 4 |
| Bezpieczeństwo | 4 | 4 | 5 |
| Konserwacja | 4 | 2 | 4 |
| Standardy programowania | 4 | 4 | 4 |
| Suma punktów | 39 | 32 | 38 |
| Znormalizowany wynik (0-1): | 0.78 | 0.64 | 0.76 |

Tabela 1 – Na powyższej tabeli przedstawiono wyniki ewaluacji dla heurystyki wielu agentów jako sędziów. Należy zauważyć, że bardzo rzadko metoda zwraca ocenę 5, czyli maksimum punktów.

W ramach przeprowadzonych eksperymentów – odnotowano, że ocena zespołu ewaluacyjnego zwykle wynosi od 32 do 39 punktów (w ostatnim z przeprowadzonych testów uzyskano ich 33) – przedstawiono to w Tabeli 1. Jednakże na tym etapie zauważono, że przeprowadzanie testów w ten sposób wymaga dużo czasu i zasobów obliczeniowych.

Zadecydowano wtedy o wdrożeniu drugiego mechanizmu ewaluacyjnego, który miał wykorzystać bibliotekę DeepEval udostępniającą metryki dedykowane systemom wieloagentowym.

W szczególności jedną z nich jest metryka G-Eval, która pozwala na ocenę przebiegu cyku rozumowania (ang. Chain Of Thoughts) takiego systemu. Szczegóły dotyczące biblioteki oraz opisywanej metryki są dostępne na stronie: <https://deepeval.com/docs/metrics-llm-evals>



Rysunek 16 - Ilustracja działania metryki G-Eval z oficjalnej strony DeepEval.

```
from deepeval.models import OllamaModel
from deepeval.test_case import LLMTestCase
from deepeval.test_case import LLMTestCaseParams

from src.crew import DevelopersCrew

def test_deepeval():
    inputs = {
        "task": "Write Pacman game in python"
    }
    final_answer = DevelopersCrew().crew().kickoff(inputs=inputs)
    actual_output = str(final_answer)

    ollama_llm = OllamaModel(model="llama3.1:8b", temperature=0.0)

    readability_metric = GEval(
        name="Readability",
        criteria="Determine if the script is easy to read, follows best coding practices, and has clear variable names with no duplicates.",
        evaluation_params=[LLMTestCaseParams.ACTUAL_OUTPUT],
        threshold=0.7,
        model=ollama_llm,
        async_mode=False
    )

    test_case = LLMTestCase(
        input=inputs["task"],
        actual_output=actual_output,
    )

    # Obowiązkowe: oblicz score
    readability_metric.measure(test_case)

    # Opcjonalnie: wyświetl score
    print(f"Readability Score: {readability_metric.score}")
    print(f"Reason: {readability_metric.reason}")

    # assert_test rzuci błąd, jeśli score < threshold
    assert_test(test_case, [readability_metric])
```

Rysunek 17 - Przykład implementacji metryki GEval w systemie CoAct. Readibility jest testem ogólnym.

```
(2 durations < 0.005s hidden. Use -vv to show these durations.)
=====
short test summary info
=====
FAILED test_deepeval.py::test_deepeval - AssertionError: Metrics: Readability [GEval] (score: 0.4, threshold: 0.7, strict: False, error: None, reason: The code does not follow best coding practices as it uses a lot of global variables, has long methods, and lacks proper error handling. The variable names are clear but duplicated in some case...)
1 failed, 10 warnings in 659.95s (0:10:59)

Test Results
=====
```

Rysunek 18 - Pierwsze wyniki ewaluacji przy pomocy metryki GEval.

Wyniki uzyskiwane w ten sposób wskazywały na pewne istotne różnice w podejściach do oceny działania systemu wieloagentowego. Metoda self-review bazowała na mechanizmie role-playowania i debat między agentami, dzięki czemu powstała ocena końcowa działania całego systemu. Ocena taka zaopatrzona jest w więcej detali, które mogą być przeoczone w przypadku zadania pojedynczego promptu modelowi językowemu.

W przypadku ewaluacji głębokiej, czyli analizy ciągu myślowego agentów (COT) pojedynczy prompt skutecznie eliminuje element losowości (dzięki mechanizmowi normalizacji prawdopodobieństw), który może się przejawić w przypadku debat agentów – dla przykładu unika się w ten sposób nieścisłości, czy dany element kodu istnieje z perspektywy konkretnego agenta, czy też nie. Różnica polega także na stopniu w jakim dane heurystyki oceniają zgodność wykonywanych czynności w stosunku do oczekiwania użytkownika. Metoda głęboka jest w stanie zapewnić węższy, lecz dokładniejszy obraz tego – analizując dogłębnie, czy oczekiwania początkowe zostały faktycznie spełnione. Natomiast ewaluacja wieloagentowa sprawdza się lepiej do badania kodu z wielu perspektyw, może to być szczególnie istotne dla oceny możliwości dalszego rozwoju tworzonego przez zespół deweloperów oprogramowania.

Porównanie wyników dla heurystyk wieloagentowej i DeepEval:

```
In terms of compliance with standards from the rules:  
* **Naming Conventions**: The variable and function names are descriptive and follow PEP 8 naming conventions. For example, 'GRID_SIZE', 'SNAKE_COLOR', and 'random_food_position' all clearly indicate their purpose.  
* **Code Organization**: The code is well-structured with clear separation of concerns between helper functions and the main game class ('SnakeGame'). This makes it easier to understand and maintain the codebase.  
* **Comments**: There are no comments in the code. While not a requirement, comments can greatly improve readability and help others understand the logic behind specific parts of the code.  
  
To further improve compliance with standards from the rules:  
1. Add docstrings to explain the purpose and behavior of each function and class.  
2. Replace magic numbers with named constants for better readability.  
3. Break down large functions into smaller ones to reduce cyclomatic complexity.  
4. Consider using type hints to indicate expected input types and return values.  
  
Overall, while there are areas where the code could be improved, it has a good foundation in terms of naming conventions, code organization, and compliance with standards from the rules.  
Total points: 29  
Average: 2.9
```

Rysunek 19 - Ewaluacja przy pomocy heurystyki wieloagentowej zwróciła wynik około 58% zgodności z wymaganiami użytkownika i zleconymi taskami.¹

```
Testing started at 01:29 ...  
Launching pytest with arguments test_deepeval.py::test_deepeval --no-header --no-summary -q in C:\Users\giantuss\PycharmProjects\coact  
===== test session starts =====  
collecting ... collected 1 item  
  
test_deepeval.py::test_deepeval PASSED [100%]  
Reason: The code adheres to best coding practices, but it does not address the issue of duplicated variable names. The variable 'FPS_START' is used in two places with dif:  
  
Running teardown with pytest sessionfinish...  
  
===== 1 passed in 12.98s =====
```

Rysunek 20 - Głęboka ewaluacja dla tego samego kodu zwraca 80% zgodności z wymaganiami użytkownika. Doprecyzowanie: Readibility uwzględnia przestrzeganie dobrych praktyk programowania, redukcję duplikatów i czytelność kodu z perspektywy użytkownika lub programisty.

¹ Doprecyzowanie – Wynik 29 / 50 punktów został niepoprawnie zsumowany dla tej iteracji testowej, po manualnym sprawdzeniu poprawny wynik powinien wynosić 32 p.

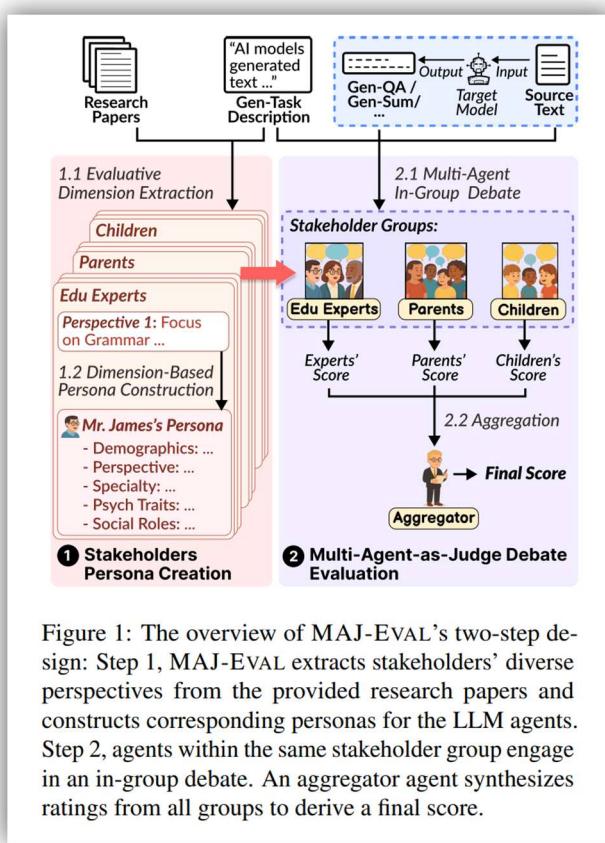
Zgodnie z oczekiwaniami, głęboka ewaluacja zwróciła bardziej optymistyczne wyniki, gdyż z reguły analizowane jest sedno samego zadania – w szczególności zgodność z treścią bądź potencjalne błędy w logice. Jednakże, nie jest to metoda bezbłędna, komentarz dotyczący podwójnego wykorzystania zmiennej FPS_START jest w tej sytuacji nie w pełni słuszna – dotyczy się ona zadeklarowania i wykorzystania zmiennej globalnej w kodzie gry. Pesymistyczne wyniki uzyskano dla analizy wieloagentowej, w przypadku niej najniżej oceniono brak dokumentacji i testów jednostkowych, poprawnie zwrócono uwagę m.in. na niewielką ilość / brak komentarzy objaśniających, oraz wykorzystanie czytelnych i zrozumiałych konwencji nazw zmiennych.

Z perspektywy testera, obie heurystyki wnoszą istotne informacje do analizy kodu. Ponadto dodatkowym czynnikiem, który powinien zostać uwzględniony w takiej analizie jest czynnik ludzki (human evaluation). W tym przypadku w ramach końcowego testu człowiek powinien dokonać ręcznej analizy wygenerowanego kodu, oceniając go pod kątem faktycznie istotnych wymagań od strony klienta jak i słuszności uwag, które zostały zwrócone przez system wieloagentowy. To dzięki tym trzem czynnikom możliwe jest uzyskanie relatywnie obiektywnej oceny gotowego programu.

Porównanie wyników z przykładem w literaturze

Aby porównać wyniki uzyskane przy pomocy metryk heurystycznych dokonano porównania z rezultatami uzyskanymi w pracy badawczej pt. „Multi-Agent-as-Judge: Aligning LLM-Agent-Based Automated Evaluation with Multi-Dimensional Human Evaluation”², której autorzy zwizualizowali wyniki ewaluacji debaty zespołu agentów. Projekt ten był zbliżony w metodycie ewaluacji do systemu CoAct, jednakże ocena odbywała się według innych wymagań użytkownika.

W tym rozdziale przedstawiono przykład systemu wieloagentowego MAJ-EVAL, który w podobny sposób analizuje odpowiedzi od agentów testerów i na podstawie ich odpowiedzi wyznacza ocenę końcową zrealizowanego projektu informatycznego.



Rysunek 21 - Wizualizacja procesu ewaluacyjnego w systemie MAJ-EVAL, który również wykorzystuje debatę agentów do oceny swojej pracy. [1]

² <https://arxiv.org/pdf/2507.21028.pdf> - Praca pt. „Multi-Agent-as-Judge: Aligning LLM-Agent-Based Automated Evaluation with Multi-Dimensional Human Evaluation” - Jiaju Chen, Yuxuan Lu, Xiaojie Wang, Huimin Zeng, Jing Huang, et. al.

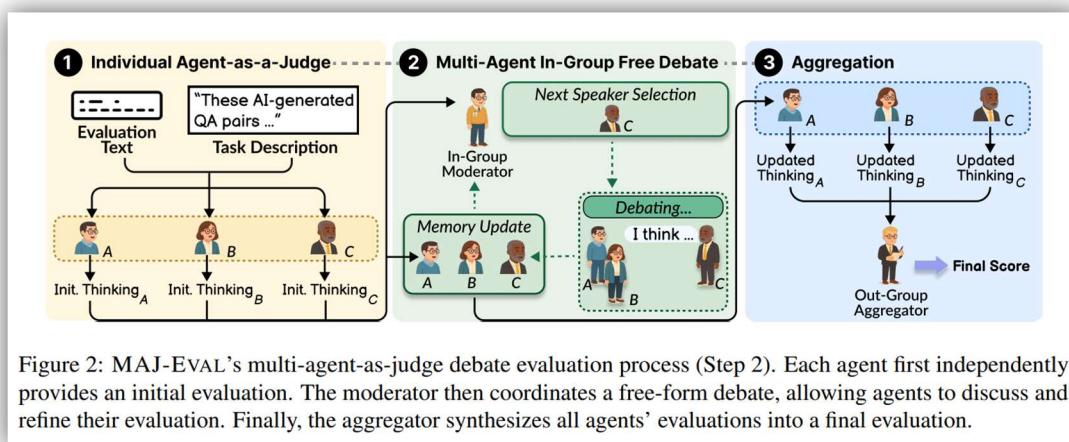


Figure 2: MAJ-EVAL's multi-agent-as-judge debate evaluation process (Step 2). Each agent first independently provides an initial evaluation. The moderator then coordinates a free-form debate, allowing agents to discuss and refine their evaluation. Finally, the aggregator synthesizes all agents' evaluations into a final evaluation.

Rysunek 22 - Wizualizacja procesów Individual-Agent-as-a-Judge oraz Multi-Agent Debate.

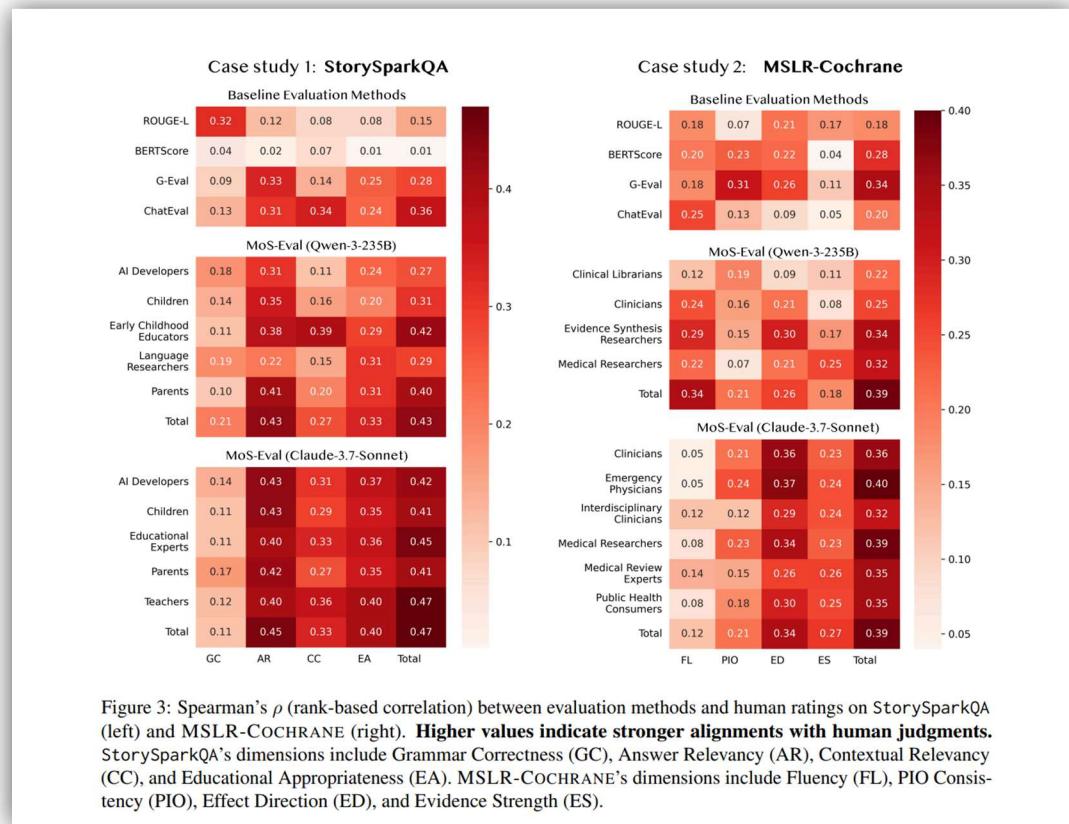


Figure 3: Spearman's ρ (rank-based correlation) between evaluation methods and human ratings on StorySparkQA (left) and MSLR-COCHRANE (right). **Higher values indicate stronger alignments with human judgments.** StorySparkQA's dimensions include Grammar Correctness (GC), Answer Relevancy (AR), Contextual Relevancy (CC), and Educational Appropriateness (EA). MSLR-COCHRANE's dimensions include Fluency (FL), PIO Consistency (PIO), Effect Direction (ED), and Evidence Strength (ES).

Rysunek 23 - Porównanie wyników do danych uzyskanych w poprzednio wspomnianych badaniach.

Instrukcja obsługi i instalacji systemu CoAct

W jaki sposób sklonować i uruchomić system wieloagentowy:

1. Na początku klonujemy repozytorium z GitHuba:
<https://github.com/Cezym/CoAct/tree/main>
2. Po sklonowaniu repozytorium musimy zainstalować narzędzie do zarządzania wirtualnymi środowiskami UV (podobne do Condy). Polecenia wykorzystywane do zainstalowania jej można znaleźć na oficjalnej stronie frameworku CrewAI: <https://docs.crewai.com/en/installation>
3. Po zainstalowaniu UV wystarczy, abyśmy wywołali w terminalu polecenie **uv sync**, aby pobrać zależności projektu CoAct.
4. Następnie uruchamiamy program przy pomocy polecenia środowiska wirtualnego: **uv run python src/main.py**
5. Dodatkowe polecenia oraz pomoc można znaleźć korzystając z flagi --help.

```
C:\Users\giantuss\PycharmProjects\coact>uv run python src/main.py --help
usage: coact [-h] [--mode {sequential,parallel}] [--evaluate] [--config-dir CONFIG_DIR] [--input INPUT] [--verbose] [--memory]

CoAct: a multi-agent system that writes code and evaluates it. Choose execution strategy with --mode (sequential|parallel). Add --evaluate to run the CodeEvaluationCrew after the main crew.

options:
-h, --help            show this help message and exit
--mode {sequential,parallel}
                      How tasks are executed, 'parallel' uses async IO.
--evaluate           Run the CodeEvaluationCrew after the main crew has finished. Useful for generating a quality report.
--config-dir CONFIG_DIR
                      Directory containing agents.yaml & tasks.yaml (default: ./config)
--input INPUT          Input prompt for DevelopersCrew.
--verbose             Print debug info.
--memory              Enable memory for crews.
```

Rysunek 24 - Lista dostępnych opcji uruchamiania systemu CoAct.

Dodatkowo w katalogu src/crew/config użytkownicy systemu mogą wprowadzać manualne zmiany w konfiguracji, w szczególności możliwe jest:

- Dodawanie lub usuwanie agentów jak i modyfikowanie ich zadań w zespole.
- Możliwe jest dodawanie i usuwanie modeli embeddingowych oraz LLM'ów. Do prezentacji wykorzystywane zostały modele lokalne, natomiast możliwe jest wykorzystanie innych, większych modeli językowych od innych firm.

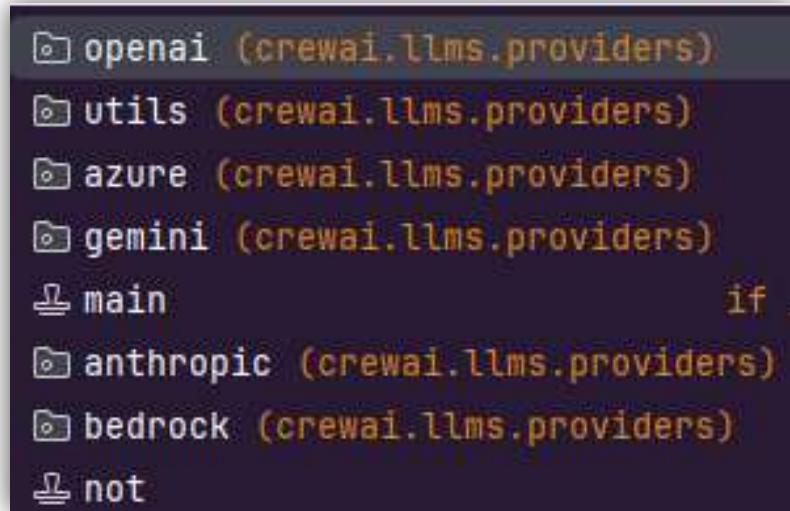
The screenshot shows a GitHub code viewer interface with the following details at the top:

- Code (selected tab)
- Blame
- 14 lines (12 loc) • 326 Bytes
- A copy icon

The code content is as follows:

```
1 local_gpt_oss_20b:  
2   provider: "openai"  
3   model: "openai/gpt-oss-20b"  
4   base_url: "http://127.0.0.1:1234/v1"  
5   api_key: "not-needed"  
6  
7 local_llama3_1_8b:  
8   model: "ollama/llama3.1:8b"  
9   api_base: "http://127.0.0.1:11434"  
10  
11 local_codegemma_7b:  
12   model: "ollama/codegemma:7b"  
13   api_base: "http://127.0.0.1:11434"  
14   temperature: 0
```

Rysunek 25 - Przykład obecnie wykorzystywanych LLM'ów w pliku `llm_providers.yaml` w konfiguracji.



Rysunek 26 - Inne opcje dla providerów modeli językowych zapewnianych w ramach integracji przez bazowy framework od CrewAI. Dostępne są między innymi LLM'y od: OpenAI, Gemini, Azure, Anthropic.

Wnioski i podsumowanie

Projekt został zakończony realizując wcześniej ustalone etapy pracy, od wstępnego ustalenia zakresu pracy, prototypu po ewaluację i porównanie wyników z kontrprzykładem w literaturze. W ramach pracy stworzony został wieloagentowy system CoAct, który może być wykorzystywany przez użytkowników do realizacji zadań z dziedziny inżynierii oprogramowania. Parametry w projekcie można zmieniać poprzez stosowanie odpowiednich opcji uruchamiania bądź zmiany w plikach konfiguracyjnych, dając użytkownikowi swobodę w zlecaniu zadań. Praca ta została udokumentowana i przedstawiona dla niniejszych przykładów programów testowych:

- Gra w Snake
- Pac-Man
- Kalkulator
- Skrypt do zewnętrznego oprogramowania

Na podstawie testów ewaluacyjnych i wynikających z nich przesłanek dostrzeżono, że stworzony system posiada potencjał do jego dalszego rozwoju. Obszarami, które można ulepszyć w przyszłości są między innymi, opracowanie strategii dzielenia projektu informatycznego na dodatkowe pliki składowe, poprawienie narzędzia wyszukiwania danych w sieci Web, udoskonalenie mechanizmu ewaluacji pracy. Projekt uświadomił również jak dużą wagę ma czynnik ludzki (human-in-the-loop) w procesie ewaluacyjnym, niektóre czynności wykonywane przez agentów LLM'owych wymagałyby dodatkowego mechanizmu weryfikacji przez użytkownika w czasie rzeczywistym, co stanowi istotny kierunek rozwoju dla przyszłych wersji systemu.
