Project ATLAS — Judyta AI Agent

Wersja dokumentu: 1.0

Status dokumentu: Wewnętrzny

System: Judyta AI Agent

Typ dokumentu: Dokumentacja techniczna

Klasyfikacja: Do użytku wewnętrznego firmy

Właściciel: [Do uzupełnienia]

Ostatnia aktualizacja: [Do uzupełnienia]

1. Cel

Niniejszy dokument opisuje architekturę, funkcjonalność, konfigurację, model bezpieczeństwa, sposób działania oraz utrzymanie agenta AI Judyta.

Dokument jest przeznaczony dla wewnętrznego personelu technicznego odpowiedzialnego za rozwój, administrację, diagnostykę oraz dalszy rozwój systemu.

Dokument opisuje zaimplementowany system i powinien być traktowany jako techniczne źródło odniesienia dla bieżącej wersji projektu.

1.1. Informacja dotycząca bezpieczeństwa

Dokument celowo nie zawiera:

kluczy API

haseł

tokenów dostępowych

prywatnych kluczy kryptograficznych

danych uwierzytelniających baz danych

danych uwierzytelniających użytkowników

innych sekretów uwierzytelniających

Wrażliwa konfiguracja musi być dostarczana za pośrednictwem zatwierdzonego mechanizmu zarządzania sekretami lub zmiennych środowiskowych.

2. Przegląd systemu

Judyta jest modułowym agentem AI zaprojektowanym do interpretowania żądań użytkownika, tworzenia planów wykonania, korzystania z zarejestrowanych narzędzi, przetwarzania ich wyników oraz generowania końcowej odpowiedzi.

System rozdziela odpowiedzialności związane z rozumowaniem na odrębne etapy:

Obsługa żądania

Planowanie

Walidacja planu

Wykonanie narzędzi

Przetwarzanie wyników

Generowanie odpowiedzi

Architektura umożliwia Judycie wykonywanie zarówno prostych operacji z użyciem pojedynczego narzędzia, jak i zadań wieloetapowych, w których wynik jednego narzędzia staje się danymi wejściowymi dla kolejnego.

3. Architektura wysokiego poziomu

Główny przepływ wykonania:

Użytkownik → Agent → Planner (LLM) → Plan / wywołania narzędzi → Parser → Executor → Registry → Narzędzia systemowe / plikowe / webowe → Wyniki narzędzi → Responder (LLM) → Odpowiedź

Centralną zasadą architektoniczną jest rozdzielenie odpowiedzialności: Planner decyduje, co należy wykonać; Executor wykonuje zaplanowane operacje; narzędzia realizują konkretne działania; Responder decyduje, w jaki sposób wynik powinien zostać przedstawiony użytkownikowi.

4. Struktura projektu

```
agent_v2/
├── main.py
├── app.py
├── config.py
├── llm.py
├── requirements.txt
│
├── core/
│   ├── agent.py
│   ├── conversation.py
│   ├── executor.py
│   ├── memory.py
│   ├── plan.py
│   ├── planner.py
│   ├── registry.py
│   ├── responder.py
│   ├── sandbox.py
│   └── tool.py
│
├── planner/
│   ├── base.py
│   ├── llm.py
│   ├── parser.py
│   ├── prompt.py
│   └── rule.py
│
├── memory/
│   └── long_term.py
│
├── tools/
│   ├── cpu.py
│   ├── disk.py
│   ├── fetch_url.py
│   ├── list_directory.py
│   ├── memory.py
│   ├── read_file.py
│   ├── recall.py
│   ├── remember.py
│   ├── system.py
│   ├── terminal.py
│   ├── web_search.py
│   └── write_file.py
```

```
│
├── tests/
└── workspace/
```

5. Główne komponenty

5.1. Agent

Implementacja: core/agent.py

Agent jest głównym komponentem orkiestrującym. Łączy Conversation, LongTermMemory, Planner, Executor, Registry oraz Responder. Agent otrzymuje żądanie użytkownika, pobiera plan od Plannera, wykonuje plan i przekazuje uzyskane informacje do Respondera. Agent rozwiązuje również odwołania do wyników wcześniejszych wywołań narzędzi.

5.2. Planner

Implementacja: planner/llm.py

Planner konwertuje żądanie zapisane w języku naturalnym na ustrukturyzowany plan wykonania. Planner nie wykonuje bezpośrednio poleceń systemu operacyjnego ani narzędzi.

5.3. Parser planu

Implementacja: planner/parser.py

Parser waliduje odpowiedź JSON Plannera przed wykonaniem. Sprawdza poprawność JSON, oczekiwaną strukturę obiektu głównego, obecność tablicy calls, poprawność każdego wywołania oraz obecność i typ nazwy narzędzia. Nieprawidłowe plany są odrzucane i nie są wykonywane.

5.4. Executor

Implementacja: core/executor.py

Executor otrzymuje zweryfikowane wywołania narzędzi i uruchamia odpowiadające im zarejestrowane narzędzia. Odpowiedzialności obejmują walidację nazwy narzędzia, pobranie narzędzia z Registry, wykonanie narzędzia oraz zwrócenie wyniku. Executor sam nie wybiera narzędzi. Wyboru dokonuje Planner.

5.5. Registry

Implementacja: core/registry.py

Registry jest centralnym katalogiem dostępnych narzędzi. Zapewnia rejestrowanie narzędzi, ochronę przed duplikatami nazw, wyszukiwanie narzędzi, sprawdzanie istnienia narzędzi, listowanie narzędzi oraz generowanie opisów narzędzi dla Plannera. Końcowy Agent rejestruje 12 narzędzi.

5.6. Responder

Implementacja: core/responder.py

Responder tworzy końcową odpowiedź dla użytkownika. Otrzymuje oryginalne żądanie, historię rozmowy, pamięć długoterminową oraz wyniki narzędzi. Responder jest instruowany, aby traktować wyniki narzędzi jako dowody i nie twierdzić, że operacja zakończyła się sukcesem, jeśli dostępne wyniki tego nie potwierdzają. Nie powinien również ujawniać wewnętrznego planowania ani informacji wrażliwych.

5.2.1. Odpowiedzialności Plannera

interpretowanie żądania użytkownika

wybór odpowiednich zarejestrowanych narzędzi

ustalanie kolejności wykonania

tworzenie argumentów narzędzi

decydowanie, kiedy wymagane są informacje zewnętrzne

korzystanie z narzędzi pamięci, gdy jest to właściwe

unikanie niepotrzebnych wywołań narzędzi

przestrzeganie ograniczeń systemowych i bezpieczeństwa

Oczekiwany format JSON: {"calls": [{"tool": "tool_name", "argument": "argument"}]}

Plan może zawierać wiele wywołań.

6. Cykl realizacji żądania

Żądanie użytkownika — żądanie trafia do Agenta.

Aktualizacja rozmowy — wiadomość użytkownika zostaje dodana do bieżącej historii.

Budowanie kontekstu — Planner otrzymuje odpowiednią historię rozmowy, pamięć długoterminową, opisy dostępnych narzędzi i bieżące żądanie.

Planowanie LLM — Planner LLM generuje plan wykonania w formacie JSON.

Walidacja planu — plan JSON jest parsowany i walidowany.

Wykonanie narzędzi — Executor wykonuje wywołania w kolejności.

Łączenie wyników — odwołania do wcześniejszych wyników są podstawiane do kolejnych argumentów.

Generowanie odpowiedzi — Responder otrzymuje wyniki i tworzy odpowiedź.

Aktualizacja rozmowy — odpowiedź asystenta zostaje dodana do bieżącej historii.

7. Wykonywanie wieloetapowe

Judyta obsługuje wykonywanie wieloetapowe.

Przykład: web_search → wynik wyszukiwania → fetch_url($previous) → treść strony.

Pozwala to łączyć kilka narzędzi w jeden workflow.

7.1. Odwołania do wyników

$previous

$result

$previous.1

$result.1

$previous.2

$result.2

{{previous}}

{{result}}

{{previous.1}}

{{result.1}}

Odwołania bez numeru wskazują najnowszy wynik. Odwołania numerowane używają indeksowania od 1. Odwołania mogą być rozwiązywane rekurencyjnie wewnątrz ciągów znaków, list, krotek i słowników.

8. Kontekst rozmowy

Implementacja: core/conversation.py

Historia rozmowy jest przechowywana w pamięci podczas działania procesu. Przechowywane są wiadomości użytkownika i asystenta.

dodawanie wiadomości użytkownika

dodawanie wiadomości asystenta

pobieranie historii

czyszczenie historii

Bieżąca implementacja nie zapewnia trwałego przechowywania historii rozmowy. Restart procesu powoduje utratę historii przechowywanej w pamięci.

9. Pamięć długoterminowa

Implementacja: memory/long_term.py

Pamięć długoterminowa jest przechowywana w: workspace/memory.json

Aktualny format opiera się na kolekcji facts. Przykład: {"facts": ["..."]}.

dodawanie faktów

pobieranie faktów

pobieranie wartości według klucza

czyszczenie pamięci

zapobieganie duplikatom

zgodność ze starszą reprezentacją pamięci

Implementacja wykorzystuje obecnie plik JSON, a nie bazę danych ani magazyn wektorowy.

10. Limity kontekstu

PromptBuilder stosuje jawne limity podczas budowania kontekstu LLM.

Kontekst Plannera:

ostatnie 6 wiadomości historii rozmowy

historia do około 5000 znaków

do 30 faktów pamięci długoterminowej

kontekst pamięci do około 3000 znaków

Kontekst Respondera obejmuje historię rozmowy, pamięć długoterminową, wyniki narzędzi i bieżące żądanie. Pojedyncze wyniki narzędzi są ograniczane przed dodaniem do kontekstu odpowiedzi.

Limity mają kontrolować rozmiar promptów i ograniczać niepotrzebny kontekst.

11. System narzędzi

Końcowy Agent rejestruje następujące 12 narzędzi:

12. Narzędzia systemowe

12.1. Uptime

Narzędzie uptime dostarcza informacji o czasie działania systemu, wykorzystując systemowe polecenie uptime.

12.2. Memory

Narzędzie memory pobiera informacje o pamięci RAM za pomocą polecenia: free -h

12.3. Disk

Narzędzie disk pobiera wykorzystanie systemów plików za pomocą polecenia: df -h

12.4. CPU

Aktualna implementacja cpu pobiera informacje o obciążeniu systemu przez uptime. Nie jest to dedykowana implementacja telemetryki CPU. Rozróżnienie to powinno być zachowane w dokumentacji technicznej i wymaganiach monitoringu.

13. Narzędzie terminal

Narzędzie terminal wykonuje polecenia powłoki za pośrednictwem systemowego interfejsu procesów.

Aktualna implementacja wykorzystuje subprocess.run(..., shell=True, timeout=120). Limit czasu polecenia wynosi 120 sekund. Narzędzie zwraca wyjście polecenia i kod zakończenia.

13.1. Tryb STANDARD

Tryb STANDARD na poziomie aplikacji blokuje wybrane polecenia o wysokim wpływie, m.in.:

sudo

su

apt

apt-get

systemctl

service

iptables

ufw

mount

umount

useradd

userdel

passwd

13.2. Tryb ADMIN

Tryb ADMIN pozwala wykonywać polecenia blokowane przez filtr aplikacyjny STANDARD.

Trybu ADMIN nie należy traktować jako oddzielnej granicy bezpieczeństwa systemu operacyjnego. Rzeczywiste uprawnienia nadal wynikają z konta systemowego, na którym działa Judyta.

14. Narzędzia plikowe

14.1. list_directory

Listuje zawartość katalogu. Aktualna implementacja nie jest ograniczona przez ten sam mechanizm safe_path(), który stosują read_file i write_file. Dostęp zależy ostatecznie od uprawnień systemu operacyjnego procesu Judyty.

14.2. read_file

Odczytuje pliki tekstowe za pośrednictwem sandboxa aplikacyjnego.

14.3. write_file

Zapisuje pliki tekstowe za pośrednictwem sandboxa aplikacyjnego.

15. Sandbox plików

Implementacja: core/sandbox.py

Aplikacja definiuje workspace/ jako kontrolowany obszar plików.

Mechanizm safe_path():

rozwiązuje wskazaną ścieżkę

traktuje ścieżki względne względem workspace

rozwiązuje przejścia ..

rozwiązuje dowiązania symboliczne

sprawdza, czy końcowa ścieżka pozostaje wewnątrz workspace

Próby wyjścia poza workspace są odrzucane.

Sandbox jest obecnie stosowany do read_file i write_file. Nie ogranicza automatycznie dowolnych poleceń powłoki wykonywanych przez terminal.

16. Narzędzia pamięci

16.1. remember

Narzędzie remember zapisuje informacje do pamięci długoterminowej. Oczekiwany format wejścia: key: value. Obsługiwane jest dodawanie nowych informacji i aktualizacja istniejących. Duplikaty nie są niepotrzebnie dodawane.

16.2. recall

Narzędzie recall pobiera pamięć długoterminową. Bez argumentu może zwrócić dostępne fakty. Z argumentem może pobrać informacje przypisane do klucza.

17. Narzędzia webowe

17.1. web_search

Narzędzie web_search zapewnia funkcję wyszukiwania w Internecie. Aktualna implementacja wykorzystuje bibliotekę ddgs.

Implementacja ogranicza liczbę zwracanych wyników, rozmiar pojedynczej treści oraz całkowity rozmiar zwracanej treści. Ograniczenia zmniejszają ilość treści zewnętrznych umieszczanej w kontekście agenta.

17.2. fetch_url

Narzędzie fetch_url pobiera wskazany zasób HTTP/HTTPS. Obsługuje typowe skompresowane odpowiedzi HTTP i próbuje ustalić prawidłowe kodowanie tekstu.

Dla HTML usuwa elementy niebędące właściwą treścią, takie jak script, style i noscript. Wyodrębniony tekst jest ograniczany rozmiarem przed przekazaniem agentowi.

18. Integracja z LLM

Implementacja: llm.py

Aktualna architektura wykorzystuje osobne role LLM dla Plannera i Respondera.

Projekt integruje się z modelami Google Gemini poprzez integrację LangChain Google Generative AI. Model jest wybierany przez konfigurację.

Dane uwierzytelniające są dostarczane przez konfigurację opartą o zmienne środowiskowe i nie mogą być umieszczane w kodzie źródłowym ani dokumentacji.

19. Konfiguracja

Konfiguracja jest ładowana przy użyciu zmiennych środowiskowych. Wrażliwa konfiguracja obejmuje m.in. klucze API.

Dokumentacja powinna opisywać nazwy zmiennych i ich przeznaczenie, ale nie ich wartości.

Przykład:

```
GEMINI_API_KEY=<secret>
MODEL=<model-name>
```

Rzeczywiste wartości muszą być dostarczane przez zatwierdzoną konfigurację wdrożeniową.

Projekt zawiera również odwołania konfiguracyjne związane z danymi uwierzytelniającymi wyszukiwania webowego; przed wdrożeniem należy je zweryfikować względem aktywnej implementacji.

20. Model bezpieczeństwa

Judyta wykorzystuje kilka mechanizmów kontroli na poziomie aplikacji.

20.1. Kontrole na poziomie promptów

Prompty Plannera i Respondera definiują zasady operacyjne, w tym unikanie ujawniania sekretów, unikanie niepotrzebnych zmian systemowych, używanie narzędzi zgodnie z przeznaczeniem, przestrzeganie ograniczeń workspace, rozróżnianie dowodów od założeń oraz unikanie twierdzeń niepopartych wynikami narzędzi.

Reguły promptów są kontrolą behawioralną i nie powinny być traktowane jako kompletna granica bezpieczeństwa.

20.2. Sandbox plików

Odczyt i zapis plików są ograniczone do workspace przez safe_path().

20.3. Ograniczenia terminala

Tryb STANDARD stosuje na poziomie aplikacji blokadę wybranych poleceń administracyjnych.

20.4. Bezpieczeństwo systemu operacyjnego

Rzeczywista granica bezpieczeństwa pozostaje po stronie systemu operacyjnego. Zalecanym modelem wdrożenia jest uruchamianie Judyty na dedykowanym koncie z minimalnym wymaganym zestawem uprawnień.

21. Zarządzanie sekretami

Sekretów nie wolno przechowywać w:

kodzie źródłowym

dokumentacji

plikach README

fixture'ach testowych

przykładowych plikach konfiguracyjnych

historii Git

Projekt powinien korzystać ze zmiennych środowiskowych lub zatwierdzonego systemu zarządzania sekretami.

W dokumentacji należy używać placeholderów: <API_KEY>, <PASSWORD>, <TOKEN>, <PRIVATE_KEY>.

Jeżeli sekret zostanie przypadkowo zapisany w repozytorium, samo usunięcie go z bieżącego pliku nie jest wystarczające. Należy uznać dane uwierzytelniające za ujawnione, przeprowadzić ich rotację oraz przejrzeć historię repozytorium.

22. Obsługa błędów

System waliduje plany przed wykonaniem. Błędy wykonania są przekazywane do warstwy orkiestracji i mogą zostać przedstawione Responderowi.

Responder ma rozróżniać operacje zakończone sukcesem, operacje zakończone błędem, niekompletne informacje oraz operacje nieobsługiwane.

Warunki limitów API, takie jak HTTP 429 / wyczerpanie zasobów, są obsługiwane jawnie.

23. CLI

Główny interfejs wiersza poleceń znajduje się w main.py.

Tryb standardowy: python main.py

Tryb administracyjny: python main.py --admin

Sesję interaktywną można zakończyć za pomocą: exit lub quit

24. Testowanie

Projekt zawiera rozbudowany zestaw testów obejmujący główne komponenty architektury.

Agent

Conversation

Planner

parser planu

Registry

Executor

pamięć długoterminowa

operacje plikowe

sandbox

łączenie wyników

ustrukturyzowane argumenty

błędy narzędzi

nieznane narzędzia

Responder

wyszukiwanie webowe

integrację Gemini

obsługę limitów API

Testy wymagające usługi Gemini są oddzielone przy użyciu markera gemini.

Polityka testowania

zainstalować dokładne zależności projektu

uruchomić kompletny lokalny zestaw testów

uruchomić zestaw testów zewnętrznego API, gdy dostępne są dane uwierzytelniające i sieć

osobno zweryfikować narzędzia wrażliwe z punktu widzenia bezpieczeństwa

zweryfikować końcową konfigurację wdrożeniową

25. Zależności

Zależności projektu są definiowane w requirements.txt.

Zestaw zależności obejmuje integrację LLM, obsługę konfiguracji, funkcjonalność HTTP/web oraz framework testowy.

Lista zależności musi być utrzymywana w zgodności z rzeczywistymi importami.

W szczególności aktywna implementacja wyszukiwania webowego importuje ddgs i zależność wymagana przez ten import powinna być jawnie zapewniona w środowisku wdrożeniowym.

26. Model operacyjny

Wdrożenie produkcyjne powinno rozdzielać:

kod aplikacji

konfigurację

sekrety

runtime workspace

logi

kopie zapasowe

Workspace powinien być traktowany jako dane aplikacji, a nie kod źródłowy. Zmiany operacyjne powinny być dokumentowane i wersjonowane.

27. Monitoring i diagnostyka

Minimalny zakres monitoringu powinien obejmować:

dostępność aplikacji

dostępność API LLM

limity API

błędy wykonywania narzędzi

nieoczekiwane zakończenie procesu

pojemność dysku

wykorzystanie pamięci

łączność sieciową dla narzędzi webowych

integralność workspace

uprawnienia konta uruchomieniowego

Preferowana kolejność diagnostyki:

odtworzyć problem

sprawdzić wyjście/logi aplikacji

zidentyfikować wadliwą warstwę architektury

przetestować dotknięty komponent niezależnie

zweryfikować konfigurację

wprowadzić najmniejszą wymaganą zmianę

powtórzyć pierwotny test

udokumentować przyczynę źródłową i rozwiązanie

28. Zasady rozwiązywania problemów

Problemy Plannera

Sprawdzić: dostępność LLM, konfigurację modelu, budowanie promptu, zwrócony JSON oraz błędy parsera.

Problemy narzędzi

Sprawdzić: rejestrację w Registry, nazwę narzędzia, argumenty, uprawnienia systemu operacyjnego oraz błędy konkretnego narzędzia.

Problemy z plikami

Sprawdzić: ścieżkę workspace, rozwiązywanie safe_path(), uprawnienia, dowiązania symboliczne oraz dostępność systemu plików.

Problemy webowe

Sprawdzić: łączność sieciową, dostępność usługi zewnętrznej, zależność ddgs, dostępność URL oraz kodowanie treści.

Problemy Respondera

Sprawdzić: wyniki narzędzi, rozmiar kontekstu, dostępność LLM, limity API oraz prompt Respondera.

29. Znane ograniczenia techniczne

terminal używa shell=True.

Standardowa ochrona terminala jest blokadą na poziomie aplikacji i nie stanowi kompletnego mechanizmu bezpieczeństwa powłoki.

list_directory nie jest obecnie chronione przez safe_path().

narzędzie cpu raportuje informacje o obciążeniu za pomocą uptime, a nie dedykowanej telemetrii CPU.

pamięć długoterminowa jest przechowywana w pliku JSON zamiast w transakcyjnej bazie danych.

historia rozmowy jest lokalna dla procesu i jest tracona po restarcie.

wykonanie planu zależy od poprawnego ustrukturyzowanego wyniku LLM.

funkcjonalność webowa zależy od usług i bibliotek zewnętrznych.

deklaracje zależności muszą pozostawać zgodne z rzeczywistymi importami.

pełna walidacja end-to-end wymaga prawidłowego środowiska uruchomieniowego i dostępu do zewnętrznych usług LLM.

30. Wytyczne rozwoju

utrzymywać rozdzielenie odpowiedzialności Plannera, Executora i Respondera

dodawać nowe możliwości jako narzędzia, gdy jest to właściwe

rejestrować narzędzia centralnie przez Registry

walidować struktury generowane przez LLM/zewnętrzne źródła przed wykonaniem

utrzymywać sekrety poza kontrolą wersji

dodawać testy dla nowych narzędzi i ścieżek wykonania

dokumentować konsekwencje bezpieczeństwa nowych możliwości

unikać niepowiązanych zmian infrastruktury podczas diagnostyki

aktualizować dokumentację po zmianach architektury lub zachowania

31. Konwencja statusu funkcji

Dokumentacja i planowanie projektu powinny rozróżniać:

IMPLEMENTED — funkcja istnieje w bieżącym kodzie i należy do aktywnej ścieżki wykonania.

PARTIALLY IMPLEMENTED — funkcja istnieje częściowo lub wymaga dalszych prac, zanim będzie można uznać ją za ukończoną.

PLANNED — funkcja jest wymaganiem przyszłym lub elementem roadmapy i nie należy do bieżącej implementacji.

Konwencja zapobiega myleniu funkcjonalności planowanej z funkcjonalnością już dostępną.

32. Zarządzanie wydaniami i zmianami

unikalny identyfikator wersji

znany commit lub tag Git

zweryfikowane zależności

przetestowaną konfigurację

udokumentowane zmiany

informację o wycofaniu/rollbacku, jeśli ma zastosowanie

Zmiany architektoniczne powinny być dokumentowane przed implementacją lub równolegle z nią.

Zmiany promptów należy traktować jako kontrolowane zmiany, ponieważ mogą istotnie wpływać na zachowanie Plannera i Respondera bez modyfikacji kodu aplikacji.

33. Bezpieczeństwo i obsługa repozytorium

Repozytorium źródłowe należy traktować jako kontrolowany zasób techniczny.

usunąć pliki zawierające sekrety

zweryfikować przykładowe konfiguracje

przeskanować drzewo robocze pod kątem danych uwierzytelniających

przejrzeć historię Git pod kątem wcześniej zapisanych sekretów

wykonać rotację danych uwierzytelniających, jeśli istnieje podejrzenie ujawnienia

zweryfikować reguły .gitignore

upewnić się, że prywatne klucze i materiały uwierzytelniające są wykluczone

34. Dalszy rozwój

Potencjalne przyszłe usprawnienia powinny być śledzone oddzielnie od bieżącej implementacji.

silniejsza izolacja procesu dla wykonywania terminala

objęcie sandboxem również listowania katalogów

dedykowane metryki CPU

trwałe przechowywanie historii rozmów

bardziej zaawansowane przechowywanie pamięci

silniejsza walidacja ustrukturyzowanych wyników

bardziej rozbudowana autoryzacja narzędzi

bogatsza obserwowalność

automatyzacja zależności i wydań

dodatkowe testy integracyjne

Pozycje te nie powinny być interpretowane jako obecnie zaimplementowane, chyba że znajdują się w oficjalnym wydaniu.

35. Załącznik — kluczowe komponenty

36. Utrzymanie dokumentu

Niniejszy dokument jest częścią wewnętrznej dokumentacji technicznej Judyty.

Powinien być przeglądany po każdej istotnej zmianie dotyczącej:

architektury systemu

dostawcy/modelu LLM

zachowania Plannera

zachowania Respondera

implementacji pamięci

zestawu narzędzi

modelu bezpieczeństwa

modelu wdrożenia

konfiguracji

uwierzytelniania

integracji zewnętrznych

Dokumentacja powinna opisywać wydany system, a nie aspiracyjną roadmapę.

Narzędzie	Zastosowanie
uptime	Informacje o czasie działania systemu
memory	Informacje o wykorzystaniu pamięci RAM
disk	Informacje o wykorzystaniu przestrzeni dyskowej
cpu	Informacje o obciążeniu systemu
terminal	Wykonywanie poleceń powłoki
list_directory	Listowanie zawartości katalogu
read_file	Odczyt pliku w ramach sandboxa
write_file	Zapis pliku w ramach sandboxa
remember	Zapisywanie informacji w pamięci długoterminowej
recall	Pobieranie informacji z pamięci długoterminowej
web_search	Wyszukiwanie informacji w Internecie
fetch_url	Pobieranie wskazanego zasobu HTTP/HTTPS

Komponent	Lokalizacja	Odpowiedzialność
Agent	core/agent.py	Główna orkiestracja
Conversation	core/conversation.py	Bieżąca historia rozmowy
Executor	core/executor.py	Wykonywanie narzędzi
Registry	core/registry.py	Katalog narzędzi

Komponent	Lokalizacja	Odpowiedzialność
Responder	core/responder.py	Generowanie końcowej odpowiedzi
Sandbox	core/sandbox.py	Walidacja ścieżek workspace
LLM Planner	planner/llm.py	Planowanie oparte na LLM
Plan Parser	planner/parser.py	Walidacja planu
Prompt Builder	planner/prompt.py	Kontekst/prompt Plannera
Long-Term Memory	memory/long_term.py	Trwałe fakty
LLM Configuration	llm.py	Klienci LLM
Application Configuration	config.py	Konfiguracja środowiskowa
CLI	main.py	Interaktywny punkt wejścia aplikacji