Ekspertowe Praktyki w Python i JSON: Od Niezawodnego Kodu po Sterowanie Modelami Językowymi

Wprowadzenie

Współczesne systemy informatyczne, od złożonych architektur mikroserwisowych po zaawansowane aplikacje oparte na sztucznej inteligencji, wymagają bezkompromisowej precyzji, wydajności i bezpieczeństwa. W tym ekosystemie, język Python i format danych JSON stanowią dwa fundamentalne filary, których synergia definiuje standardy nowoczesnego inżynierii oprogramowania. Niniejszy raport przedstawia wyczerpującą syntezę najlepszych praktyk, zaawansowanych wzorców i strategii dla Pythona i JSON, kulminując w analizie nowatorskich technik sterowania Wielkimi Modelami Językowymi (LLM) w celu uzyskania weryfikowalnych i ustrukturyzowanych wyników. Analiza ta jest skierowana do doświadczonych inżynierów, architektów i specjalistów AI, dostarczając skondensowanej, gotowej do wdrożenia wiedzy niezbędnej do budowy systemów produkcyjnych najwyższej klasy.

Ewolucja Praktyk w Języku Python: Wydajność, Jakość i Niezawodność

Ekosystem Pythona przeszedł w ostatnich latach transformację, odchodząc od rozproszonego zestawu narzędzi na rzecz zintegrowanych, wysokowydajnych rozwiązań. Ta ewolucja, napędzana przez potrzebę większej szybkości, prostoty i niezawodności, ustanowiła nowy standard dla profesjonalnego rozwoju oprogramowania.

Fundamenty Nowoczesnego Kodu: Zintegrowany Zestaw Narzędzi

Obserwuje się fundamentalną zmianę w ekosystemie narzędzi deweloperskich Pythona. Narzędzia takie jak ruff i uv, napisane w Rust, oferują wydajność o rzędy wielkości wyższą niż ich poprzednicy, co bezpośrednio przekłada się na produktywność deweloperów i szybkość procesów ciągłej integracji.[1] Ta ewolucja to nie tylko optymalizacja; sygnalizuje ona strategiczne przejście w kierunku konsolidacji i wykorzystania języków kompilowanych do wzmocnienia środowiska deweloperskiego Pythona.

- **Ruff: Konsolidacja Lintingu i Formatowania**: Ruff jest narzędziem, które zrewolucjonizowało analizę statyczną i formatowanie kodu Python. Jego wydajność, od 10 do 100 razy większa niż tradycyjnych linterów, wynika z implementacji w Rust.[1] Co ważniejsze,
  ruff zastępuje całą gamę narzędzi, takich jak Flake8 (wraz z dziesiątkami wtyczek), isort, black, pydocstyle i autoflake.[2] Ta konsolidacja znacząco upraszcza konfigurację projektu – zamiast wielu plików konfiguracyjnych, wystarczy jedna sekcja w pliku pyproject.toml. Umożliwia to egzekwowanie spójnego stylu w całym zespole przy minimalnym narzucie konfiguracyjnym i drastycznie skraca czas działania potoków CI/CD.
- **uv: Nowa Generacja Zarządzania Zależnościami**: uv, również stworzony przez twórców ruff, stanowi przełom w zarządzaniu zależnościami i środowiskami wirtualnymi. Jest on od 10 do 100 razy szybszy niż pip i pip-tools, a jego funkcjonalność pozwala zastąpić pip, pip-tools oraz virtualenv jednym, spójnym narzędziem.[1] Kluczową cechą
  uv jest wykorzystanie globalnego, współdzielonego bufora (cache) dla pobranych pakietów, co znacznie oszczędza miejsce na dysku i przyspiesza tworzenie nowych środowisk.[1]
- **Mypy: Bezpieczeństwo Typów w Standardzie**: Statyczna analiza typów, zdefiniowana w PEP 484, stała się nieodłącznym elementem profesjonalnego programowania w Pythonie. Mypy jest de facto standardem w tej dziedzinie, pozwalając na wykrywanie błędów związanych z typami jeszcze przed uruchomieniem kodu.[3] Integracja
  mypy w procesie CI/CD jest kluczową praktyką, która poprawia niezawodność, czytelność i łatwość utrzymania kodu, zwłaszcza w dużych i złożonych projektach.[1]
- **Zarządzanie Projektem i Wersjami**: Należy unikać używania systemowej instalacji Pythona do celów deweloperskich. Zamiast tego, zaleca się stosowanie menedżerów wersji, takich jak pyenv, które pozwalają na łatwe przełączanie się między różnymi wersjami interpretera dla poszczególnych projektów.[4] Projekty powinny być zarządzane za pomocą dedykowanych narzędzi, takich jak
  Poetry, PDM czy Hatch, które automatyzują tworzenie struktury projektu, zarządzanie zależnościami i budowanie pakietów.[4]

Kategoria	Narzędzia	Nowoczesny	Kluczowa Przewaga

	Tradycyjne	Standard	
Linting i Formatowanie	Flake8, Black, isort, pydocstyle	ruff	Wydajność (Rust), konsolidacja konfiguracji, szybkość CI/CD
Zarządzanie Zależnościami	pip, pip-tools, virtualenv	uv	Wydajność (Rust), globalny cache, zunifikowane API
Walidacja Danych	Ręczna walidacja, Marshmallow	Pydantic	Integracja z typami, generowanie schematów, ekosystem (FastAPI)
Analiza Statyczna	Brak lub opcjonalna	mypy (zintegrowany z CI)	Wczesne wykrywanie błędów, bezpieczeństwo typów, lepsza refaktoryzacja

Kontrakty Danych z Pydantic: Definiowanie Pojedynczego Źródła Prawdy

Rola Pydantic w nowoczesnym stosie technologicznym Pythona wyewoluowała daleko poza prostą walidację danych. Stał się on centralnym elementem architektury, który umożliwia tworzenie ścisłych, samopisujących się kontraktów danych. Te kontrakty działają jak "pojedyncze źródło prawdy" (single source of truth), łącząc system typów Pythona, schematy JSON dla API oraz konfigurację aplikacji w spójną i niezawodną całość.

Pydantic wykorzystuje natywne adnotacje typów Pythona do definiowania modeli danych, które automatycznie walidują, parsują i konwertują dane wejściowe.[1] Ta cecha jest fundamentem dla frameworków takich jak FastAPI, gdzie modele Pydantic są używane do automatycznego generowania specyfikacji OpenAPI (w tym schematów JSON) oraz do

walidacji danych w żądaniach i odpowiedziach.[1] Ten sam model Pydantic może być również użyty do zarządzania konfiguracją aplikacji, wczytując ustawienia ze zmiennych środowiskowych lub plików.

Połączenie Pydantic z mypy za pomocą dedykowanej wtyczki (pydantic.mypy) tworzy potężny mechanizm zapewnienia jakości. Wtyczka ta rozszerza możliwości mypy, dostarczając szczegółowej analizy statycznej specyficznej dla modeli Pydantic.[5] Umożliwia ona m.in. weryfikację poprawności wywołań konstruktora

__init__, egzekwowanie niezmienności modeli (frozen=True) oraz sprawdzanie zgodności typów dla wartości domyślnych.[5] Aby osiągnąć maksymalne bezpieczeństwo typów, kluczowa jest odpowiednia konfiguracja wtyczki, na przykład poprzez ustawienie

init_forbid_extra = True, aby zapobiec przekazywaniu nieoczekiwanych pól do konstruktora, oraz init_typed = True, aby wymusić ścisłą zgodność typów argumentów.[5]

W efekcie, jedna definicja modelu Pydantic pełni cztery kluczowe funkcje: (1) walidację danych w czasie wykonania, (2) statyczną analizę typów na etapie dewelopmentu, (3) definicję kontraktu API (serializacja/deserializacja) oraz (4) zarządzanie konfiguracją aplikacji. Taka centralizacja drastycznie redukuje duplikację kodu, eliminuje niespójności między warstwami systemu i promuje metodologię contract-first, która jest fundamentem solidnych architektur rozproszonych.

Wzorce Stylistyczne i Idiomatyczne (PEP 8 i więcej)

Pisanie "pythonicznego" kodu, zgodnego z duchem języka i wytycznymi społeczności, jest kluczowe dla tworzenia oprogramowania, które jest czytelne, łatwe w utrzymaniu i wydajne. PEP 8, oficjalny przewodnik po stylu dla kodu Python, stanowi fundament, ale najlepsze praktyki wykraczają poza jego ramy, obejmując również idiomatyczne wykorzystanie struktur języka.[6]

- **Układ Kodu i Białe Znaki**: Zgodnie z PEP 8, funkcje i klasy najwyższego poziomu powinny być otoczone dwiema pustymi liniami, aby wizualnie oddzielić od siebie niezależne bloki funkcjonalne. Metody wewnątrz klas powinny być rozdzielone jedną pustą linią, co sygnalizuje ich wzajemne powiązanie. Wewnątrz funkcji, puste linie mogą być używane oszczędnie do oddzielania logicznych kroków, co znacząco poprawia czytelność złożonych algorytmów.[6] Należy konsekwentnie używać spacji wokół operatorów binarnych (np. przypisania, porównania, logicznych), co ułatwia wizualne parsowanie wyrażeń.[6]
- **Konwencje Nazewnictwa**: Stosowanie spójnych konwencji nazewnictwa jest jednym z

najważniejszych aspektów czytelności. PEP 8 zaleca snake_case dla funkcji, metod i zmiennych, PascalCase (lub CapWords) dla klas, oraz UPPER_SNAKE_CASE dla stałych.[6] Zrozumienie konwencji dotyczących podkreśleń (np. _single_leading_underscore dla użytku wewnętrznego, __double_leading_underscore do manglingu nazw w klasach) jest również istotne dla zaawansowanych programistów.[7]

- **Komentarze i Docstringi**: Komentarze powinny być zwięzłe i wyjaśniać "dlaczego" kod robi to, co robi, a nie "co" robi. Komentarze blokowe muszą być wcięte na tym samym poziomie co kod, który opisują, a każda linia powinna zaczynać się od # i pojedynczej spacji.[6] Docstringi (ciągi dokumentacyjne) są niezbędne do dokumentowania modułów, klas, funkcji i metod, i powinny być pisane zgodnie ze standardem PEP 257.
- **Idiomy Językowe**: Wykorzystanie wbudowanych funkcji i idiomów Pythona prowadzi do bardziej zwięzłego i wydajnego kodu. Na przykład, należy preferować użycie funkcji enumerate() w pętlach, które wymagają licznika, zamiast ręcznego inkrementowania zmiennej.[7] Do sprawdzania przynależności do kolekcji należy używać operatorów in i not in, które są czytelniejsze i często bardziej wydajne (zwłaszcza dla set i dict) niż iterowanie po elementach.[7] Efektywne łączenie stringów, na przykład za pomocą metody str.join() zamiast wielokrotnej konkatenacji z operatorem +, jest kolejnym przykładem idiomatycznej i wydajnej praktyki.[7]

Wzorce i Listy Kontrolne w Pythonie

Poniżej przedstawiono skondensowane listy kontrolne i wzorce, które stanowią podsumowanie kluczowych praktyk w zakresie wydajności, bezpieczeństwa, utrzymania i testowania kodu Python.

Wzorce Projektowe

Wzorzec	Uzasadnienie (Why)	Implementacja (How)	Zastosowanie (When)	Pułapki (Pitfalls)	Snippet (Python)	Referencje
Użycie Pydantic jako Kontraktu Danych	Zapewnia pojedyncze źródło prawdy dla walidacji,	Definiuj modele dziedziczące po pydantic.BaseMod	Wszędzie tam, gdzie dane są wymieniane: API,	Nadmierne poleganie na automatycznej	`from pydantic import BaseModel\n\nclass	None = None`

	serializacji i analizy statycznej.	el z adnotacjami typów.	pliki konfiguracyjne, komunikacja między serwisami.	konwersji typów może ukryć błędy; używaj trybu strict.	User(BaseModel):\n id: int\n name: str\n email: str	
Zintegrowany Tooling (Ruff + Mypy)	Konsoliduje i przyspiesza procesy lintingu, formatowania i analizy typów.	Skonfiguruj ruff i mypy w pliku pyproject.toml i zintegruj z CI/CD.	W każdym nowoczesnym projekcie Python, od początku jego istnienia.	Początkowa konfiguracja mypy w istniejącym projekcie może być czasochłonna.	[tool.ruff]\nline-length = 88\n\n[tool.mypy]\nplugins = ["pydantic.mypy"]\ndisallow_untyped_defs = true	[1]
Preferowanie Kompozycji nad Dziedziczeniem	Zwiększa elastyczność i unika problemów związanych ze złożonymi hierarchiami klas.	Twórz klasy, które zawierają instancje innych klas jako atrybuty.	Gdy obiekt "ma" (has-a) inną funkcjonalność, a nie "jest" (is-a) jej specjalizacją.	Może prowadzić do większej liczby obiektów "przekazujących" wywołania (boilerplate).	class Engine:\n def start(self):...\n\nclass Car:\n def __init__(self):\n self.engine = Engine()	[7]
Obsługa Błędów w Stylu EAFP	Upraszcza kod, czyniąc go bardziej	Używaj bloków try...except do obsługi	W sytuacjach, gdzie błędy są rzadkie, a	Może ukrywać błędy programistyczne,	try:\n value = my_dict[key]\nexcept	[7]

	czytelnym i optymistycznym.	wyjątkowych sytuacji, zamiast wielu warunków if.	ścieżka "szczęśliwa" jest dominująca.	jeśli blok except jest zbyt ogólny (np. except Exception:).	KeyError: \n value = "default"	
Użycie enumerate dla Pętli z Licznikiem	Zwiększa czytelność i eliminuje potrzebę ręcznego zarządzania indeksem.	Użyj for index, value in enumerate(iterable): w pętlach.	Zawsze, gdy potrzebujesz zarówno wartości, jak i jej indeksu w pętli.	Brak; jest to standardowy idiom Pythona.	names =\nfor i, name in enumerate(names):\n print(f"{i+1}: {name}")	[7]

Listy Kontrolne

- **Wydajność**:
  - Używaj str.join() zamiast + do łączenia wielu stringów.
  - Preferuj set i dict do sprawdzania przynależności (operacja O(1)).
  - Profiluj kod za pomocą cProfile lub py-spy, aby zidentyfikować wąskie gardła przed optymalizacją.
  - Wykorzystuj generatory i yield do przetwarzania dużych zbiorów danych bez ładowania ich do pamięci.
  - Rozważ użycie bibliotek takich jak Polars dla operacji na danych, które wykorzystują leniwe ewaluacje (lazy evaluation).[7]
- **Bezpieczeństwo**:
  - Waliduj wszystkie dane wejściowe za pomocą Pydantic lub JSON Schema.
  - Nigdy nie osadzaj sekretów (kluczy API, haseł) bezpośrednio w kodzie; używaj zmiennych środowiskowych lub systemów zarządzania sekretami.[8]
  - Używaj bezpiecznych parserów dla formatów zewnętrznych (np. defusedxml dla XML).
  - Loguj zdarzenia związane z bezpieczeństwem, ale unikaj logowania danych wrażliwych.
  - Regularnie aktualizuj zależności, aby unikać znanych podatności.
- **Utrzymanie**:
  - Stosuj się do PEP 8, używając ruff format do automatyzacji.
  - Stosuj adnotacje typów we wszystkich nowych funkcjach i stopniowo dodawaj je do

istniejącego kodu.
    - Dziel kod na małe, spójne moduły o jasno zdefiniowanych odpowiedzialnościach.
    - Pisz zwięzłe i użyteczne docstringi dla wszystkich publicznych API.
    - Refaktoryzuj kod regularnie, aby upraszczać logikę i usuwać duplikacje.
- **Testy**:
    - Dąż do wysokiego pokrycia kodu testami, używając narzędzi jak pytest-cov.
    - Pisz testy jednostkowe dla izolowanych komponentów i testy integracyjne dla interakcji między nimi.[7]
    - Używaj pytest fixtures do zarządzania stanem testów w sposób czysty i reużywalny.
    - Stosuj parametryzację testów (@pytest.mark.parametrize), aby testować wiele przypadków brzegowych za pomocą jednej funkcji testowej.
    - Integruj uruchamianie testów i analizę statyczną w potokach CI/CD.

JSON jako Kontrakt Danych: Standardy, Bezpieczeństwo i Wersjonowanie

JSON, z uwagi na swoją prostotę i czytelność, stał się uniwersalnym językiem wymiany danych w internecie. Jednak jego efektywne wykorzystanie w systemach produkcyjnych wymaga dyscypliny i stosowania standardów, które przekształcają go z prostego formatu serializacji w solidny kontrakt danych.

JSON Schema jako Fundament Niezawodności i Bezpieczeństwa

JSON Schema jest kluczowym, choć często niedocenianym, elementem w budowie niezawodnych i bezpiecznych systemów rozproszonych. Jego rola wykracza poza prostą walidację; stanowi on formalny, maszynowo weryfikowalny kontrakt, który definiuje dozwoloną strukturę, typy i ograniczenia dla dokumentów JSON.[9] W architekturze mikroserwisowej, gdzie dziesiątki lub setki usług komunikują się ze sobą, JSON Schema staje się gwarantem spójności i pierwszą linią obrony przed nieprawidłowymi danymi.

Zastosowanie JSON Schema jako mechanizmu kontroli bezpieczeństwa jest jednym z jego najpotężniejszych aspektów. Definiując ścisły, oparty na liście dozwolonych pól (allowlist) kontrakt na brzegu API lub aplikacji, można skutecznie zneutralizować całe klasy zagrożeń z listy OWASP API Security Top 10, takich jak "Mass Assignment" (API3:2023) czy "Injection" (API8:2023).[11] Na przykład, atak polegający na próbie aktualizacji profilu użytkownika z

dodatkowym, nieautoryzowanym polem

{"isAdmin": true} zostanie natychmiast odrzucony, jeśli schemat nie definiuje takiego pola i używa dyrektywy "additionalProperties": false.[11] Podobnie, egzekwowanie formatów (np.

"format": "date-time") i wzorców (np. "pattern": "^[a-zA-Z0-9-]{1,50}$") zapobiega wielu formom ataków typu injection.[14] To przekształca JSON Schema z narzędzia walidacyjnego w potężny, deklaratywny mechanizm bezpieczeństwa, który implementuje politykę "domyślnie odmawiaj" (

implicit deny) na poziomie struktury danych.

Zagrożenie (OWASP API Security Top 10:2023)	Opis	Mitygacja za pomocą JSON Schema
API3: Broken Object Property Level Authorization	Atakujący manipuluje właściwościami obiektu, do których nie powinien mieć dostępu (np. isAdmin: true).	Użyj "properties" do jawnego zdefiniowania dozwolonych pól. Ustaw "additionalProperties": false, aby odrzucać wszystkie nieznane pola.
API8: Security Misconfiguration (Injection)	Wstrzykiwanie złośliwych danych (SQLi, NoSQLi, command injection) przez pola tekstowe.	Użyj ścisłych walidacji "pattern" (regex) dla wszystkich stringów. Używaj predefiniowanych "format" (np. "email", "uuid"). Stosuj "enum" dla pól o ograniczonej liczbie wartości.
API6: Unrestricted Access to Sensitive Business Flows	Atakujący wysyła dane o nieprawidłowym typie lub strukturze, aby wywołać nieoczekiwane błędy lub ścieżki w kodzie.	Stosuj ścisłą walidację typów ("type": "integer") oraz ograniczenia ("minimum", "maximum") dla wszystkich pól.

JSON Schema jest również fundamentem specyfikacji OpenAPI (wersje 2.0, 3.0 i 3.1), co czyni go de facto standardem w dokumentowaniu i egzekwowaniu kontraktów dla API webowych.[15]

Zarządzanie Ewolucją Schematów: Wersjonowanie i Kompatybilność

W dynamicznych systemach zmiany w schematach danych są nieuniknione. Kluczem do zarządzania tą ewolucją bez zakłócania działania usług jest przyjęcie zdyscyplinowanej strategii wersjonowania. Zasady Wersjonowania Semantycznego (SemVer), powszechnie stosowane w oprogramowaniu, można z powodzeniem zaadaptować do schematów JSON.[16]

- **MAJOR (np. v2.0.0)**: Wersja główna jest inkrementowana, gdy wprowadzana jest **zmiana łamiąca kompatybilność wsteczną (breaking change)**. Przykłady takich zmian to:
  - Usunięcie wymaganego pola.
  - Zmiana typu danych istniejącego pola (np. z string na integer).
  - Uczynienie opcjonalnego pola wymaganym.
  - Dodanie nowej, restrykcyjnej walidacji, która unieważnia wcześniej poprawne dane.[16]
- **MINOR (np. v1.1.0)**: Wersja poboczna jest inkrementowana dla **nowych funkcjonalności, które są wstecznie kompatybilne (non-breaking change)**. Oznacza to, że konsumenci przygotowani na wersję v1.0.0 mogą bezpiecznie przetwarzać dane zgodne z v1.1.0, ignorując nowe elementy. Przykłady:
  - Dodanie nowego, opcjonalnego pola.
  - Dodanie nowej wartości do istniejącego enum.
  - Rozszerzenie wzorca pattern, aby akceptował więcej wartości.[16]
- **PATCH (np. v1.0.1)**: Wersja poprawkowa jest przeznaczona na **wstecznie kompatybilne poprawki błędów**, które nie zmieniają API, np. korekta w opisie (description) pola.

Najlepszą praktyką, która minimalizuje potrzebę wprowadzania zmian łamiących, jest projektowanie schematów w sposób rozszerzalny. Zamiast usuwać lub zmieniać nazwy istniejących pól, preferuje się dodawanie nowych i oznaczanie starych jako przestarzałe (deprecated).[17] Wersję schematu można umieścić bezpośrednio w dokumencie JSON (np. pole

schema_version: "v1.1.0") lub, co jest bardziej eleganckie, w URI identyfikatora schematu ($id: "https://example.com/schemas/user/v1.1.0.json").[18]

Konwencje i Pułapki Semantyczne: null vs. Brak Pola

Precyzyjne wyrażanie "braku wartości" jest jednym z najczęstszych źródeł nieporozumień i błędów w systemach opartych na JSON. Istnieje fundamentalna różnica semantyczna między polem, którego wartość jest jawnie ustawiona na null, a polem, które w ogóle nie jest obecne

w obiekcie JSON.[20]

- **Jawne null ({"key": null}):** Oznacza, że wartość pola jest znana i celowo jest pusta lub nie ma zastosowania. Jest to jawna informacja.
- **Brak klucza ({}):** W większości języków programowania (np. undefined w JavaScript) oznacza to, że informacja o tym polu nie została dostarczona.

To rozróżnienie jest krytyczne, zwłaszcza w kontekście częściowych aktualizacji zasobów (np. za pomocą metody HTTP PATCH). Jeśli użytkownik wysyła żądanie {"description": null}, intencją jest wyczyszczenie opisu. Jeśli jednak pole description jest po prostu pominięte w żądaniu, intencją jest pozostawienie go bez zmian.[20] Ignorowanie tej różnicy po stronie serwera prowadzi do nieprzewidywalnego zachowania API.

Najlepsze praktyki dotyczące null i typów:

- **Używaj null jawnie:** Do reprezentowania braku wartości dla istniejącego atrybutu.[21]
- **Puste kolekcje jako :** Zawsze zwracaj pustą tablicę () zamiast null dla kolekcji, które nie mają elementów. Pozwala to uniknąć po stronie klienta konieczności sprawdzania if (items!== null) przed iteracją.[21]
- **Unikaj "null" jako stringa:** Wartość {"key": "null"} to string, a nie wartość null, co jest częstym źródłem błędów.[21]
- **Spójność kluczy:** Używaj spójnej konwencji nazewnictwa kluczy w całym API (np. camelCase lub snake_case).

Wzorce i Listy Kontrolne dla JSON

Najlepsze Praktyki Projektowania JSON

Zasada (Rule)	Uzasadnienie (Rationale)	Przykład (JSON)	Antywzorzec (Anti-pattern)	Referencje
Używaj JSON Schema do walidacji	Egzekwuje kontrakt danych, poprawia bezpieczeństwo i niezawodność.	{"$schema": "...", "type": "object", "properties": {...}, "required": [...]}	Przetwarzanie JSON bez walidacji struktury.	[9]

Wersjonuj schematy semantycznie	Komunikuje charakter zmian (breaking vs. non-breaking).	"$id": "https://api.com/schemas/user/v2.0.json"	Brak wersjonowania lub losowe wersjonowanie.	[16]
Preferuj dodawanie pól nad modyfikacją	Zapewnia kompatybilność wsteczną i ułatwia ewolucję API.	// v1: {"name": "A"}\n// v2: {"name": "A", "fullName": "A B"}	// v1: {"name": "A"}\n// v2: {"fullName": "A B"}	[17]
Rozróżniaj null od braku pola	Umożliwia precyzyjne operacje, zwłaszcza częściowe aktualizacje (PATCH).	{"field": null} (wyczyść pole) vs. {} (nie ruszaj pola).	Traktowanie braku pola tak samo jak null.	[20]
Zwracaj puste kolekcje jako ``	Upraszcza logikę klienta, unika błędów NullPointerException.	{"items":}	{"items": null}	[21]
Używaj spójnej konwencji kluczy	Poprawia czytelność i przewidywalność API.	{"userId": 1, "firstName": "John"} (camelCase)	Mieszanie snake_case i camelCase.	[6]

Listy Kontrolne

- **Wskazówki dotyczące schematów (Schema Tips)**:
  - Zawsze dołączaj $schema i $id do swoich schematów, aby zadeklarować wersję specyfikacji i unikalny identyfikator.[23]
  - Bądź jak najbardziej precyzyjny: używaj pattern dla stringów, minimum/maximum dla liczb, minItems/maxItems dla tablic.
  - Używaj "additionalProperties": false jako domyślnej, bezpiecznej opcji, aby odrzucać nieznane pola.
  - Komponuj złożone schematy za pomocą $ref, allOf, anyOf, oneOf, aby promować reużywalność.

    - Dokumentuj swoje schematy za pomocą pól title i description.
- **Wskazówki dotyczące bezpieczeństwa (Security Tips)**:
    - Waliduj wszystkie dane wejściowe po stronie serwera za pomocą schematu; nie ufaj walidacji po stronie klienta.[14]
    - Ogranicz rozmiar przychodzących dokumentów JSON, aby zapobiec atakom typu Denial of Service.
    - Używaj HTTPS dla całej komunikacji API.[11]
    - Nie umieszczaj danych wrażliwych (kluczy API, tokenów) w odpowiedziach JSON; używaj bezpiecznych nagłówków HTTP.[11]
    - Stosuj odpowiednie nagłówki bezpieczeństwa, takie jak Content-Security-Policy i X-Content-Type-Options: nosniff, nawet dla odpowiedzi API.[11]

Sterowanie LLM: Strukturyzowane Wyjścia i Weryfikowalne Rozumowanie

Interakcja z Wielkimi Modelami Językowymi (LLM) przechodzi ewolucję od formy sztuki, opartej na kreatywnym formułowaniu promptów w języku naturalnym, do zdyscyplinowanej inżynierii. Ta "industrializacja" promptingu jest napędzana potrzebą niezawodności, weryfikowalności i integracji LLM jako przewidywalnych komponentów w większych systemach. W tym nowym paradygmacie, JSON i pseudokod stają się kluczowymi narzędziami do precyzyjnego sterowania zachowaniem modeli.

Paradygmat Dekodowania z Ograniczeniami (Constrained Decoding)

Standardowe LLM generują tekst token po tokenie, wybierając następny token na podstawie rozkładu prawdopodobieństwa. Ten proces, choć skuteczny w generowaniu płynnego języka, jest z natury probabilistyczny i nie gwarantuje przestrzegania żadnej określonej struktury. W zastosowaniach produkcyjnych, gdzie wynik musi być parsowalny i zgodny z kontraktem (np. odpowiedź API), swobodne generowanie tekstu jest nieakceptowalne.[24]

Rozwiązaniem tego problemu jest **dekodowanie z ograniczeniami (constrained decoding)**. Jest to technika, która interweniuje w procesie generowania na każdym kroku. Zamiast pozwalać modelowi na wybór dowolnego tokenu z jego słownika, nakładana jest maska, która dopuszcza tylko te tokeny, które są w danym momencie zgodne z predefiniowaną gramatyką

lub schematem.[26]

W praktyce, JSON Schema stało się de facto standardem branżowym do definiowania tych ograniczeń.[26] Kiedy LLM ma wygenerować obiekt JSON, proces dekodowania na bieżąco sprawdza, czy następny token może być częścią poprawnego dokumentu JSON zgodnego z podanym schematem. Na przykład, jeśli schemat oczekuje klucza

"name", a model wygenerował {"na, dekoder dopuści tylko tokeny, które mogą prowadzić do ukończenia tego klucza (np. m, me). Po wygenerowaniu {"name": ", jeśli schemat definiuje typ wartości jako integer, dekoder zablokuje wszystkie tokeny, które nie są cyframi. Ten proces gwarantuje, że końcowy wynik będzie nie tylko poprawnym składniowo dokumentem JSON, ale także będzie w 100% zgodny z dostarczonym schematem. Frameworki takie jak Guidance, Outlines, a także natywne API od OpenAI i Google, implementują tę technikę, co znacząco podnosi niezawodność LLM w zadaniach wymagających ustrukturyzowanych danych.[26]

Pseudokod jako "Łańcuch Postępowania" (Chain of Conduct)

Technika "Chain of Thought" (CoT) pokazała, że instruowanie LLM, aby "myślał krok po kroku", poprawia jego zdolności rozumowania. Jednak rozumowanie w języku naturalnym pozostaje często niejednoznaczne, rozwlekłe i trudne do automatycznej weryfikacji. Kolejnym krokiem ewolucyjnym jest zastąpienie języka naturalnego bardziej ustrukturyzowanym medium – pseudokodem.

Użycie pseudokodu to coś więcej niż tylko ulepszony "łańcuch myśli"; ustanawia ono **"łańcuch postępowania" (Chain of Conduct)**. Podczas gdy CoT pokazuje, *co* model myśli, CoC dyktuje, *jak* model musi postępować. Ta różnica jest kluczowa dla budowy audytowalnych i weryfikowalnych systemów AI.

Badania wykazują, że promptowanie za pomocą pseudokodu prowadzi do znacznie lepszych wyników niż instrukcje w języku naturalnym, ponieważ pseudokod jest z natury bardziej precyzyjny, zwięzły i mniej podatny na błędne interpretacje.[28] Zmusza on model do dekompozycji problemu na logiczne, sekwencyjne kroki, definiowania zmiennych i stosowania struktur kontrolnych, co tworzy weryfikowalną ścieżkę rozumowania.[30]

Ta ścieżka rozumowania, wyrażona w pseudokodzie, nie jest przeznaczona tylko dla ludzkiego czytelnika. Może być ona analizowana programistycznie w celu sprawdzenia logicznej spójności, poprawności użycia zmiennych i tego, czy ostateczny wniosek wynika z przesłanek zdefiniowanych w krokach pseudokodu. Techniki takie jak "Hint of Thought" (HoT) wykorzystują ten mechanizm, prosząc model o generowanie odpowiedzi na pod-pytania w formie pseudokodu, co czyni jego proces myślowy transparentnym i weryfikowalnym.[31] W ten

sposób pseudokod przekształca rozumowanie LLM z nieprzejrzystego procesu wewnętrznego w audytowalny "łańcuch postępowania", co jest fundamentalnym krokiem w kierunku budowy godnej zaufania sztucznej inteligencji.

Architektura Promptów Opartych na JSON i Kontrakty Wyjściowe

Najwyższym poziomem inżynierii promptów jest traktowanie samego promptu jako ustrukturyzowanego dokumentu – formalnej specyfikacji zadania. Zamiast luźnego bloku tekstu, cały prompt jest formatowany jako obiekt JSON, który zawiera jasno zdefiniowane sekcje.

Standardowy wzorzec dla takiego promptu obejmuje klucze:

- task: Zwięzły opis zadania do wykonania.
- data: Obiekt lub tablica zawierająca dane wejściowe, na których model ma operować.
- constraints: Lista reguł lub ograniczeń, których model musi przestrzegać.
- output_schema: Definicja JSON Schema dla oczekiwanego wyniku.

Dołączenie output_schema bezpośrednio do promptu jest kluczowe. Daje to modelowi jasny obraz celu i jest niezbędnym elementem dla mechanizmów dekodowania z ograniczeniami.[26] Po otrzymaniu odpowiedzi od LLM, kluczowym krokiem jest natychmiastowa walidacja wygenerowanego JSON-a względem tego samego

output_schema. Taka pętla zwrotna pozwala na automatyczną ocenę poprawności wyniku. W przypadku błędu walidacji, system może automatycznie ponowić zapytanie, dołączając do nowego promptu informację o błędach, co umożliwia modelowi samokorektę.[24]

Ten cykl – **specyfikacja (prompt JSON) -> generowanie z ograniczeniami -> walidacja (JSON Schema)** – przekształca interakcję z LLM w przewidywalny i niezawodny proces inżynieryjny, umożliwiając budowę złożonych, wieloetapowych systemów, w których LLM pełni rolę deterministycznego komponentu przetwarzającego dane.

Wzorce i Listy Kontrolne dla Interakcji z LLM

Wzorce Promptów

Nazwa Wzorca	Zastosowanie (When)	Szablon (JSON)	Pseudokod (CoC)	Przykład Testu	Referencje
Zadanie ze Schematem Wyjściowym	Gdy wymagany jest ściśle zdefiniowany, strukturalny wynik.	{\n "task": "Extract user info from text.",\n "data": {\"text\": \"...\"},\n "output_schema": {\n "type": "object",\n "properties": {\n "name": {"type": "string"},\n "age": {"type": "integer"}\n },\n "required": ["name"]\n }\n}	1. Read input text.\n2. Identify name entity.\n3. Identify age entity.\n4. Construct JSON object matching output_schema.	Input: "John is 30."\nExpected: {"name": "John", "age": 30}	[26]
Rozumowanie z Łańcuchem Postępowania	Dla złożonych problemów wymagających dekompozycji i weryfikowalnych kroków.	{\n "task": "Solve the logic puzzle.",\n "puzzle_data": {...},\n "instructions":,\n "output_schema": {"type": "object",...}\n}	PLAN:\n solve(puzzle: Puzzle) -> Answer\n [Validation] Ensure puzzle data is not empty.\n Analyze premise A.\n Analyze	Input: Logic puzzle\nExpected: Correct answer + verifiable pseudocode trace.	[30]

			premise B.\n Deduce conclusion from A and B.\n [Edge Cases] Handle contradictions.\n [Complexity] O(N)		
Pętla z Samokorektą	Gdy pierwsza próba generacji może się nie powaść i wymagana jest iteracyjna poprawa.	{\n "task": "Generate a valid configuration file.",\n "requirements": {...},\n "previous_attempt": {\n "output": {...},\n "validation_errors": ["Field 'x' is missing"]\n },\n "output_schema": {...}\n}	1. Review previous output and validation errors.\n2. Identify the root cause of the error (e.g., missing field 'x').\n3. Modify the output to correct the error.\n4. Re-validate against the schema.	Input: Invalid config + error message\n Expected: Valid config.	[24]

Listy Kontrolne

- **Zasady "Lintingu" dla Promptów**:
  - Bądź precyzyjny i jednoznaczny; unikaj języka potocznego i metafor.
  - Jawnie określ format wyjściowy, najlepiej dołączając JSON Schema.
  - Używaj przykładów (few-shot prompting) dla złożonych lub niestandardowych formatów.
  - Oddzielaj instrukcje od danych wejściowych za pomocą wyraźnych separatorów lub struktury JSON.
  - Testuj prompty pod kątem przypadków brzegowych i nieoczekiwanych danych

wejściowych.
- **Zasady Walidacji Wyjścia**:
  - Zawsze waliduj wyjście LLM względem schematu przed dalszym przetwarzaniem.
  - Implementuj logikę ponawiania prób w przypadku błędów walidacji.
  - Loguj błędy walidacji, aby monitorować wydajność modelu i identyfikować problematyczne prompty.
  - Sprawdzaj nie tylko poprawność strukturalną (schema), ale także poprawność semantyczną (czy dane mają sens w kontekście biznesowym).
  - Odrzucaj odpowiedzi, które, mimo że są zgodne ze schematem, zawierają halucynacje lub dane niepoparte źródłem.

Wnioski

Analiza najlepszych praktyk w ekosystemach Pythona i JSON ujawnia wyraźny trend w kierunku inżynierii oprogramowania opartej na **dyscyplinie, automatyzacji i formalnych kontraktach**. Ewolucja narzędzi deweloperskich w Pythonie, zdominowana przez wysokowydajne, skonsolidowane rozwiązania takie jak ruff i uv, demonstruje dążenie do maksymalizacji produktywności i niezawodności w cyklu życia oprogramowania. Jednocześnie, Pydantic i statyczna analiza typów z mypy ugruntowały paradygmat, w którym dane są traktowane jako obywatele pierwszej kategorii, z jasno zdefiniowanymi, weryfikowalnymi na wielu poziomach kontraktami.

W dziedzinie wymiany danych, JSON Schema przestało być jedynie narzędziem walidacyjnym, a stało się fundamentalnym mechanizmem bezpieczeństwa i egzekwowania kontraktów w architekturach rozproszonych. Przyjęcie zdyscyplinowanych strategii wersjonowania, opartych na zasadach SemVer, oraz precyzyjne zarządzanie semantyką wartości null są kluczowe dla budowy elastycznych i odpornych na błędy systemów.

Te zasady formalizacji i kontraktów osiągają swój najpełniejszy wyraz w interakcji z Wielkimi Modelami Językowymi. Przejście od nieprecyzyjnych promptów w języku naturalnym do ustrukturyzowanych specyfikacji w formacie JSON, w połączeniu z dekodowaniem z ograniczeniami i weryfikowalnym rozumowaniem opartym na pseudokodzie, stanowi zmianę paradygmatu. Przekształca ono LLM z probabilistycznego generatora tekstu w przewidywalny, programowalny komponent, który można bezpiecznie integrować w krytycznych systemach produkcyjnych. Opanowanie tych trzech, wzajemnie powiązanych dziedzin, jest obecnie niezbędne dla każdego inżyniera i architekta dążącego do tworzenia oprogramowania najwyższej jakości.

Referencje

ID	Tytuł	URL	Autor/Wydawca	Data Publikacji	Data Dostępu		
	[7]		Python Best Practices for More Pythonic Code	https://realpython.com/tutorials/best-practices/	Real Python	2025-08-26	2025-08-26
	[6]		How to Write Beautiful Python Code With PEP 8	https://realpython.com/python-pep8/	Real Python	2025-01-12	2025-08-26
	[1]		A Modern Python Toolkit: Pydantic, Ruff, MyPy, and UV	https://dev.to/devasservice/a-modern-python-toolkit-pydantic-ruff-mypy-and-uv-4b2f	Developer Service	2024-06-19	2025-08-26
	[9]		JSON	https://j	JSON	2022-0	2025-0

			Schema Validation	son-schema.org/draft/2020-12/json-schema-validation	Schema Org	6-16	8-26
	10		JSON Subschema Checking	https://arxiv.org/pdf/1911.12651	G. R. M. Timm, et al.	2019-11-28	2025-08-26
	15		A Tutorial on JSON Schema	https://www.openproceedings.org/2025/conf/edbt/paper-T3.pdf	D. Calvanese, et al.	2025-03-25	2025-08-26
	34		JSON Schema Core	https://json-schema.org/draft/2020-12/json-schema-core	JSON Schema Org	2022-06-16	2025-08-26
	26		JSONSchema Bench: A Rigorous Bench	https://arxiv.org/html/2501.10868v1	S. Geng, et al.	2025-01-24	2025-08-26

			mark of Structured Outputs for Language Models				
	24		Schema Bench: A Comprehensive Benchmark for Structured JSON Generation	https://arxiv.org/html/2502.18878v1	Y. Xia, et al.	2025-02-28	2025-08-26
	30		Code as a Medium for Reasoning	https://arxiv.org/html/2502.19411v1	Z. Chen, et al.	2025-02-28	2025-08-26
	28		Fine-tuning Language Models with Just a Hint of Pseudo-code	https://arxiv.org/html/2505.18011v1	A. Murthy, et al.	2025-05-28	2025-08-26

	25		StructuredRAG: JSON Response Formatting with Large Language Models	https://arxiv.org/abs/2408.11061	C. Shorten, et al.	2024-08-07	2025-08-26
	29		Can We Edit Models? A Study of Editing Large Language Models with Pseudo-Code Instructions	https://arxiv.org/abs/2305.11790	N. Mishra, et al.	2023-05-19	2025-08-26
	3		Mypy documentation	https://mypy.readthedocs.io/	Mypy team	2025-08-26	2025-08-26
	4		Modern Good Practices for Python Development	https://www.stuartellis.name/articles/python-modern	Stuart Ellis	2024-05-26	2025-08-26

				-practices/			
	5		Mypy plugin for Pydantic	https://docs.pydantic.dev/latest/integrations/mypy/	Pydantic	2025-08-26	2025-08-26
	2		Ruff: An extremely fast Python linter, written in Rust.	https://github.com/astral-sh/ruff	Astral	2025-08-26	2025-08-26
	1		A Modern Python Toolkit: Pydantic, Ruff, MyPy, and UV	https://dev.to/devasservice/a-modern-python-toolkit-pydantic-ruff-mypy-and-uv-4b2f	Developer Service	2024-06-19	2025-08-26
	13		Mitigate OWASP API threats	https://learn.microsoft.com/en-us/azure/api-management/mitigate-o	Microsoft	2024-07-29	2025-08-26

				wasp-api-threats			
	11		REST Security Cheat Sheet	https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html	OWASP Foundation	2024-07-16	2025-08-26
	12		OWASP API Security Top Ten - Essential Takeaways for JSON Developers	https://moldstud.com/articles/p-owasp-api-security-top-ten-essential-takeaways-for-json-developers	MoldStud	2024-01-01	2025-08-26
	8		Secrets Management Cheat Sheet	https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat	OWASP Foundation	2024-07-16	2025-08-26

				_Sheet.html			
	14		Input Validation Cheat Sheet	https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html	OWASP Foundation	2024-07-16	2025-08-26
	23		Understanding JSON Schema - Basics	https://json-schema.org/understanding-json-schema/basics	JSON Schema Org	2025-08-26	2025-08-26
	16		Semantic Versioning 2.0.0	https://semver.org/	Tom Preston-Werner	2013-06-19	2025-08-26
	18		Schema Versioning Tutorial	https://developer.couchbase.com/tutorial-schema-versioning/?learningPath=learn/j	Couchbase	2025-08-26	2025-08-26

				son-document-management-guide			
	[19]		Is there a standard for specifying a version for JSON schema?	https://stackoverflow.com/questions/61077293/is-there-a-standard-for-specifying-a-version-for-json-schema	Stack Overflow	2020-04-20	2025-08-26
	[17]		What is the best way to handle versioning using JSON protocol?	https://stackoverflow.com/questions/10042742/what-is-the-best-way-to-handle-versioning-using-json-protocol	Stack Overflow	2012-04-04	2025-08-26
	[21]		Representing null in	https://stackoverflow.co	Stack Overflow	2014-01-14	2025-08-26

			JSON	m/questions/21120999/representing-null-in-json			
	20		How to determine if a JSON key has been set to null or not provided	https://www.calhoun.io/how-to-determine-if-a-json-key-has-been-set-to-null-or-not-provided/	Jon Calhoun	2025-08-26	2025-08-26
	22		Should JSON include null values?	https://stackoverflow.com/questions/11003424/should-json-include-null-values	Stack Overflow	2012-06-12	2025-08-26
	24		Schema Bench: A Comprehensive Benchmark	https://arxiv.org/html/2502.18878v1	Y. Xia, et al.	2025-02-28	2025-08-26

			for Structured JSON Generation				
	26		JSONSchemaBench: A Rigorous Benchmark of Structured Outputs for Language Models	https://arxiv.org/html/2501.10868v1	S. Geng, et al.	2025-01-24	2025-08-26
	27		YieldLang: A Coroutine-based DSL Generation Framework	https://arxiv.org/abs/2404.05499	Z. Wang, et al.	2024-04-08	2025-08-26
	31		Hint of Thought prompting: an explainable and	https://arxiv.org/html/2305.11461v7	Z. Ye, et al.	2023-05-18	2025-08-26

			zero-shot approach to reasoning tasks with LLMs				
	[35]		CodeAgents: A Token-Efficient Framework for Codified Multi-Agent Reasoning in LLMs	https://arxiv.org/html/2507.03254v1	Y. Wen, et al.	2025-07-04	2025-08-26
	[33]		How Does LLM Reasoning Work for Code? A Survey and a Call to Action	https://arxiv.org/html/2506.13932v1	A. Sharma, et al.	2025-06-16	2025-08-26
	[32]		Hint of Though	https://arxiv.or	Z. Ye, et al.	2023-05-18	2025-08-26

			t prompting: an explainable and zero-shot approach to reasoning tasks with LLMs	g/html/2305.11461v6			
	7		Python Best Practices for More Pythonic Code	https://realpython.com/tutorials/best-practices/	Real Python	2025-08-26	2025-08-26
	9		JSON Schema Validation	https://json-schema.org/draft/2020-12/json-schema-validation	JSON Schema Org	2022-06-16	2025-08-26
	5		Mypy plugin for Pydantic	https://docs.pydantic.dev/latest/integrations/mypy/	Pydantic	2025-08-26	2025-08-26

	11		REST Security Cheat Sheet	https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html	OWASP Foundation	2024-07-16	2025-08-26
	16		Semantic Versioning 2.0.0	https://semver.org/	Tom Preston-Werner	2013-06-19	2025-08-26
	21		Representing null in JSON	https://stackoverflow.com/questions/21120999/representing-null-in-json	Stack Overflow	2014-01-14	2025-08-26

Cytowane prace

1. A Modern Python Toolkit: Pydantic, Ruff, MyPy, and UV - DEV Community, otwierano: sierpnia 26, 2025, https://dev.to/devasservice/a-modern-python-toolkit-pydantic-ruff-mypy-and-uv-4b2f
2. astral-sh/ruff: An extremely fast Python linter and code formatter, written in Rust. - GitHub, otwierano: sierpnia 26, 2025, https://github.com/astral-sh/ruff
3. mypy 1.17.1 documentation, otwierano: sierpnia 26, 2025, https://mypy.readthedocs.io/
4. Modern Good Practices for Python Development - Stuart Ellis, otwierano: sierpnia 26, 2025, https://www.stuartellis.name/articles/python-modern-practices/
5. Mypy - Pydantic, otwierano: sierpnia 26, 2025, https://docs.pydantic.dev/latest/integrations/mypy/

6. How to Write Beautiful Python Code With PEP 8 – Real Python, otwierano: sierpnia 26, 2025, https://realpython.com/python-pep8/
7. Python Best Practices – Real Python, otwierano: sierpnia 26, 2025, https://realpython.com/tutorials/best-practices/
8. Secrets Management - OWASP Cheat Sheet Series, otwierano: sierpnia 26, 2025, https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html
9. JSON Schema Validation: A Vocabulary for Structural Validation of ..., otwierano: sierpnia 26, 2025, https://json-schema.org/draft/2020-12/json-schema-validation
10. Type Safety with JSON Subschema - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/pdf/1911.12651
11. REST Security - OWASP Cheat Sheet Series, otwierano: sierpnia 26, 2025, https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html
12. OWASP API Security Top Ten - Essential Takeaways for JSON Developers - MoldStud, otwierano: sierpnia 26, 2025, https://moldstud.com/articles/p-owasp-api-security-top-ten-essential-takeaways-for-json-developers
13. Mitigate OWASP API security top 10 in Azure API Management | Microsoft Learn, otwierano: sierpnia 26, 2025, https://learn.microsoft.com/en-us/azure/api-management/mitigate-owasp-api-threats
14. Input Validation - OWASP Cheat Sheet Series, otwierano: sierpnia 26, 2025, https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html
15. Everything You Always Wanted to Know About JSON Schema (But Were Afraid to Ask) - OpenProceedings.org, otwierano: sierpnia 26, 2025, https://www.openproceedings.org/2025/conf/edbt/paper-T3.pdf
16. Semantic Versioning 2.0.0 | Semantic Versioning, otwierano: sierpnia 26, 2025, https://semver.org/
17. What is the best way to handle versioning using JSON protocol? - Stack Overflow, otwierano: sierpnia 26, 2025, https://stackoverflow.com/questions/10042742/what-is-the-best-way-to-handle-versioning-using-json-protocol
18. Learning Path - Schema Versioning | Couchbase Developer Portal, otwierano: sierpnia 26, 2025, https://developer.couchbase.com/tutorial-schema-versioning/?learningPath=learn/json-document-management-guide
19. Is there a standard for specifying a version for json schema - Stack Overflow, otwierano: sierpnia 26, 2025, https://stackoverflow.com/questions/61077293/is-there-a-standard-for-specifying-a-version-for-json-schema
20. How to determine if a JSON key has been set to null or not provided - Calhoun.io, otwierano: sierpnia 26, 2025, https://www.calhoun.io/how-to-determine-if-a-json-key-has-been-set-to-null-or-not-provided/

21. Representing null in JSON - Stack Overflow, otwierano: sierpnia 26, 2025, https://stackoverflow.com/questions/21120999/representing-null-in-json
22. Should JSON include null values - javascript - Stack Overflow, otwierano: sierpnia 26, 2025, https://stackoverflow.com/questions/11003424/should-json-include-null-values
23. The basics - JSON Schema, otwierano: sierpnia 26, 2025, https://json-schema.org/understanding-json-schema/basics
24. arxiv.org, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2502.18878v1
25. [2408.11061] StructuredRAG: JSON Response Formatting with Large Language Models, otwierano: sierpnia 26, 2025, https://arxiv.org/abs/2408.11061
26. Generating Structured Outputs from Language Models: Benchmark and Studies - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2501.10868v1
27. [2404.05499] Guiding Large Language Models to Generate Computer-Parsable Content, otwierano: sierpnia 26, 2025, https://arxiv.org/abs/2404.05499
28. Training with Pseudo-Code for Instruction Following - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2505.18011v1
29. [2305.11790] Prompting with Pseudo-Code Instructions - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/abs/2305.11790
30. Code to Think, Think to Code: A Survey on Code-Enhanced Reasoning and Reasoning-Driven Code Intelligence in LLMs - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2502.19411v1
31. Hint of Thought prompting: an explainable and zero-shot approach to reasoning tasks with LLMs - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2305.11461v7
32. Hint of Thought prompting: an explainable and zero-shot approach to reasoning tasks with LLMs - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2305.11461v6
33. How Does LLM Reasoning Work for Code? A Survey and a Call to Action - arXiv, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2506.13932v1
34. A Media Type for Describing JSON Documents - JSON Schema, otwierano: sierpnia 26, 2025, https://json-schema.org/draft/2020-12/json-schema-core
35. CodeAgents: A Token-Efficient Framework for Codified Multi-Agent Reasoning in LLMs, otwierano: sierpnia 26, 2025, https://arxiv.org/html/2507.03254v1