Technologie Anthropic dla projektu Jarvis

Analiza dokumentacji i rekomendacje implementacyjne

Data: 17.01.2026 **Autor:** Research session _jarvis **Status:** Do analizy i przemyślenia
Kontekst: Projekt _jarvis — centralny asystent AI z ambicją pełnej autonomii

---

Streszczenie

Niniejszy artykuł podsumowuje dogłębną analizę dokumentacji Anthropic pod kątem realizacji wizji projektu _jarvis. Zidentyfikowano kluczowe technologie dla każdej z 4 faz rozwoju: MCP (Model Context Protocol), Claude API z Tool Use, Agent SDK oraz techniki optymalizacji kosztów.

Główne wnioski: 1. **FastMCP** to optymalna droga do MVP (Faza 1) 2. **Tool Runner (beta)** automatyzuje cykl tool use — idealny dla integracji Faza 2 3. **Prompt caching** oferuje 90% oszczędność na powtarzalnych kontekstach 4. **Architektura orchestrator + subagents** jest potwierdzoną best practice dla systemów autonomicznych

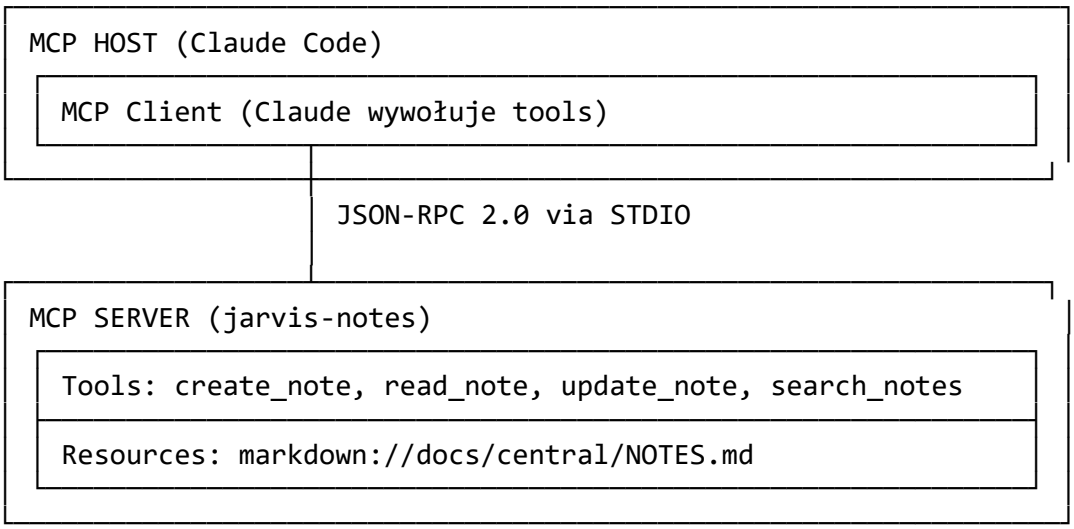
---

1. Model Context Protocol (MCP) — Fundament Fazy 1

Czym jest MCP?

Model Context Protocol to otwarty standard komunikacji między aplikacjami hostującymi (jak Claude Code, Claude Desktop) a zewnętrznymi serwerami narzędzi. Pozwala rozszerzać możliwości Claude o własne funkcje bez modyfikacji samego modelu.

Architektura komunikacji



FastMCP — rekomendowane podejście

FastMCP to wysokopoziomowy framework Python, który drastycznie upraszcza tworzenie serwerów MCP:

```python
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("jarvis-notes")

@mcp.tool()
async def create_note(title: str, content: str, file: str, tags: list[str] = No
ne) -> str:
    """Create a new note entry in markdown file.

    Args:
        title: Note title (can include tags like @_jarvis)
        content: Note content
        file: Target markdown file (relative to project root)
        tags: Optional list of tags for categorization

    Returns:
        Entry ID and confirmation message
    """
    # Implementacja CRUD na markdown
    entry_id = await handler.create_entry(file, title, content, tags)
    return f"Note created: {entry_id} in {file}"
```

Kluczowe zasady implementacji

Aspekt	Wymaganie
Transport	STDIO dla Claude Code (najszybszy, bezpośredni)
Protokół	JSON-RPC 2.0 — asynchroniczny, stateful
Logging	MUSI być do stderr/file, **NIGDY** stdout (psuje JSON-RPC!)
Walidacja	Pydantic — automatyczne schematy JSON
Błędy	Zwracaj `isError: true` + opis dla self-correction

Zastosowanie dla Jarvis

Serwer `jarvis-notes` powinien obsługiwać: - **create_note** — “zanotuj: ...” → NOTES.md z tagiem @projekt - **create_todo** — “do zrobienia: ...” → TODO.md - **search_knowledge** — “co wiemy o X?” → przeszukanie KNOWLEDGE.md - **add_journal** — “refleksja: ...” → JOURNAL.md z datownikiem

---

2. Claude API — Tool Use i Streaming (Faza 2)

Tool Use — definiowanie narzędzi

Każde narzędzie wymaga precyzyjnej definicji:

```json
{
  "name": "send_gmail",
  "description": "Wysyła email przez Gmail API. Obsługuje: odbiorcę, CC, BCC, z
ałączniki. Używaj gdy użytkownik mówi: 'wyślij email', 'napisz do'. NIE obsługu
je harmonogramowania.",
  "input_schema": {
    "type": "object",
```

```
    "properties": {
      "to": {"type": "string", "description": "Adres email odbiorcy"},
      "subject": {"type": "string", "description": "Temat wiadomości, max 100 z
naków"},
      "body": {"type": "string", "description": "Treść HTML lub tekst"}
    },
    "required": ["to", "subject", "body"]
  }
}
```

Kluczowa zasada: Opis narzędzia = minimum 3-4 zdania: CO robi, KIEDY, OGRANICZENIA.

Tool Runner (beta) — automatyzacja cyklu

Nowość w API Anthropic — Tool Runner automatycznie zarządza: - Iteracjami aż do zakończenia (end_turn) - Parallel tool calls - Error handling - Message history

```python
from anthropic import beta_tool

@beta_tool
def send_gmail(to: str, subject: str, body: str) -> str:
    """Wysyła email."""
    return f"Email sent: ID_12345"

@beta_tool
def create_calendar_event(title: str, start_time: str, end_time: str) -> str:
    """Tworzy zdarzenie w kalendarzu."""
    return f"Event created: event_789"

# Tool Runner obsługuje automatycznie cały cykl
runner = client.beta.messages.tool_runner(
    model="claude-sonnet-4-5",
    tools=[send_gmail, create_calendar_event],
    messages=[{
        "role": "user",
        "content": "Wyślij email do John'a i stwórz spotkanie na jutro o 10"
    }]
)

# Claude sam zdecyduje o kolejności i równoległości wywołań
final = runner.until_done()
```

Streaming — odpowiedzi na żywo

Dla interfejsów jak Telegram — streaming daje lepszy UX:

```python
async def send_to_telegram_streaming(user_id, prompt):
    """Wysyła odpowiedź na żywo do Telegrama"""
    message_id = await send_empty_telegram_message(user_id)
    buffer = ""

    with client.messages.stream(
        model="claude-sonnet-4-5",
        messages=[{"role": "user", "content": prompt}]
    ) as stream:
        for text in stream.text_stream:
```

```
            buffer += text
            if len(buffer) > 200:
                await edit_telegram_message(user_id, message_id, buffer)

        await edit_telegram_message(user_id, message_id, buffer)
```

Batching — 50% oszczędności

Dla operacji nie wymagających real-time (nocne digesty, klasyfikacja emaili):

- **Deadline:** 24h (większość < 1h)
- **Rabat:** 50% na wszystkie tokeny
- **Use case:** Analiza 100 emaili, weekly digest, raportowanie

---

3. Agent SDK — Architektura autonomii (Faza 3-4)

Wzorzec Orchestrator + Subagents

Rekomendowana architektura dla Jarvisa:

```
ORCHESTRATOR (_jarvis-core)
├─ Role: Globalne planowanie, routing, state management
├─ Permissions: Wąskie ("read and route")
└─ Kontekst: CLAUDE.md + MEMORY.md

├── SUBAGENT: notes-manager
│   ├─ Wejście: "zanotuj: ..."
│   ├─ Wyjście: wpis w JOURNAL.md / NOTES.md
│   └─ Izolacja: własne okno kontekstu
│
├── SUBAGENT: todo-coordinator
│   ├─ Wejście: "do zrobienia: ..."
│   ├─ Wyjście: wpis w TODO.md z tagiem @projekt
│   └─ Izolacja: własne okno kontekstu
│
├── SUBAGENT: email-handler
│   ├─ Wejście: "sprawdź maile o X"
│   ├─ Wyjście: podsumowanie + akcje
│   └─ Izolacja: własne okno kontekstu
│
└── SUBAGENT: knowledge-retriever
    ├─ Wejście: "co wiemy o X?"
    ├─ Wyjście: przeszukanie bazy wiedzy
    └─ Izolacja: własne okno kontekstu
```

Poziomy autonomii

Poziom	Nazwa	Opis	Faza Jarvis
L1	Human-in-the-loop	Agent prosi o pozwolenie na każdą zmianę	Faza 1-2
L2	Human-on-the-loop	Auto dla safe, pyta dla risky	Faza 2-3

Poziom	Nazwa	Opis	Faza Jarvis
L3	Autonomous	Agent działa sam, human monitoruje via logs	Faza 3
L4	Self-improver	Jarvis buduje Jarvisa 2.0	Faza 4

Session Management — pamięć między sesjami

Problem: Claude nie pamięta między sesjami. Rozwiązanie: Externalizacja pamięci do plików.

```
docs/central/
├── MEMORY.md          # Co agent zapamiętał o użytkowniku/systemie
├── JOURNAL.md         # Historia decyzji i zdarzeń
├── KNOWLEDGE.md       # Baza wiedzy
└── claude-progress.txt # Log postępu (czytaj na starcie sesji)
```

Praktyka: 1. Na koniec sesji → agent pisze do MEMORY.md 2. Na starcie sesji → Initializer Agent czyta MEMORY.md + progress.txt 3. Stare konteksty → archiwizacja do JOURNAL.md

Context Window Management

Claude 4.5 ma 200K tokenów kontekstu. Strategie zarządzania:

Strategia	Opis	Use case
Core bucket	NIGDY nie kompresuj	CLAUDE_RULES.md, krytyczne instrukcje
Working bucket	Podsumuj jeśli >20 bloków	Aktualna rozmowa, stan zadania
Transient bucket	Usuń po użyciu	Tymczasowe notatki, tool results

---

4. Optymalizacja kosztów

Prompt Caching — 90% oszczędności

Oznacz statyczną zawartość `cache_control`:

```
response = client.messages.create(
    model="claude-sonnet-4-5",
    system=[
        {
            "type": "text",
            "text": open("docs/central/CLAUDE_RULES.md").read(),
            "cache_control": {"type": "ephemeral"}  # Cache 5 min
        },
        {
            "type": "text",
            "text": open("docs/central/KNOWLEDGE.md").read(),
            "cache_control": {"type": "ephemeral"}
        }
    ],
```

```
    messages=[...]
)
```

Ekonomia: - Cache write: $3.75/MTok (1.25x cena) - Cache read: $0.30/MTok (0.1x cena) - **ROI:** Po ~13 requestach cache zwraca koszt

Guardrails — bezpieczeństwo wielowarstwowe

Warstwa 1: Haiku pre-screening - Szybki (5-10x tańszy) filtr przed głównym modelem - Klasyfikacja: safe/unsafe

Warstwa 2: System prompt constraints - Jawne reguły w systemowym prompcie - “Never bypass user permissions”

Warstwa 3: Tool use constraints - Whitelist dozwolonych operacji w settings.json - Agent nie może wywoływać operacji poza listą

---

5. Mapa technologii na fazy projektu

Faza	Cel	Technologia Anthropic	Implementacja
F1: FUNDAMENT	Mówię → zapisuje	MCP + FastMCP	jarvis-notes server
F2: ZASIĘG	Telegram, Gmail, Calendar	Tool Use + Streaming + Tool Runner	Integracje Google, bot TG
F3: MÓZG	Głos, RAG, kontekst	Agent SDK + Orchestrator	Subagents, Vector DB
F4: AUTONOMIA	Mówię raz → działa zawsze	Multi-AI Router + Self-improvement	L3-L4 autonomia

---

6. Rekomendacje natychmiastowe

Do wdrożenia teraz (koszt: $0)

1. **Prompt caching** — dodaj `cache_control` do centralnych plików
2. **Struktura MEMORY.md** — zdefiniuj format wpisów sesji
3. **claude-progress.txt** — log postępu dla continuity

Do wdrożenia w Fazie 1 MVP

1. **FastMCP server jarvis-notes** — CRUD na markdown
2. **Integracja z Claude Code** — `.mcp.json` w repozytorium

3. **Test sprawczości:** “zanotuj: X” → X jest w NOTES.md

Do zaplanowania (Faza 2+)

1. **Tool Runner** dla Gmail/Calendar integracji
2. **Telegram bot** ze streamingiem
3. **Batch API** dla nocnych digestów

---

7. Pytania otwarte

1. **MCP vs bezpośrednie API** — czy MCP jest optymalny dla wszystkich integracji, czy tylko dla Claude Code?
2. **Subagent granularity** — jak drobno dzielić subagenty? Jeden per funkcja czy per domena?
3. **Memory persistence** — MEMORY.md vs dedykowana baza (SQLite, PostgreSQL)?
4. **Koszty w skali** — przy 100+ requestów/dzień, jaki model (Haiku vs Sonnet vs Opus) dla jakiego zadania?
5. **Self-improvement (L4)** — jak bezpiecznie pozwolić agentowi modyfikować własny kod?

Źródła

- Model Context Protocol — Build Server
- Anthropic API — Tool Use Implementation
- Anthropic API — Streaming
- Anthropic — Agent SDK Overview
- Anthropic — Prompt Caching
- Anthropic — Batch Processing
- Building Effective Harnesses for Long-Running Agents

Status dokumentu: Do analizy i przemyślenia **Następny krok:** Decyzja o kolejności implementacji **Powiązane:** ROADMAP.md, CONCEPT.md