ChatGPT

Architektura agenta Zordon (Lolek)

1. Długotrwałe operacje i niezawodność

Aby obejść ograniczenia czasowe funkcji serverless na Vercelu (np. 10 s na planie darmowym, 300 s na Pro [1]), należy podzielić zadania na mniejsze kroki i użyć zewnętrznego mechanizmu orkiestracji. Przykładowo Inngest pozwala definiować wieloetapowe funkcje backgroundowe wyzwalane na podstawie zdarzeń – każda faza przepływu (send email, oczekiwanie itp.) wykonuje się osobno i jest automatycznie wznawiana przy błędach [2] [3]. Trigger.dev w wersji v3 z kolei uruchamia kod we własnych kontenerach z użyciem CRIU (checkpoint/restore), co praktycznie eliminuje limity czasu wykonania [4]. Można też skorzystać z kolejkowania (np. Upstash QStash) – rozbić duże zadanie na wiele równoległych zadań lambda, każde mieszczące się w limicie [5]. Wreszcie do prostych, periodycznych zadań można użyć Vercel Cron: działa on przez wywołanie API (`/api/cron`) na zadany harmonogram [6], ale pamiętać trzeba, że każde uruchomienie nadal podlega limitowi czasu funkcji (domyślnie kilkanaście sekund) [1].

Orkiestracja zadań Inngest: Zdarzenie „user.signup” wyzwala wieloetapową funkcję, która wysyła e-mail i oczekuje – każdy krok jest niezależnie wznawiany w razie błędu [2] [3]. Dzięki takiemu podejściu złożone procesy można bezpiecznie prowadzić poza główną ścieżką synchroniczną aplikacji.

Rekomendowane rozwiązania (2025): zamiast próbować znacznie wydłużać timeout (np. `maxDuration` w Next.js), lepiej użyć narzędzi “durable functions”. Inngest czy Trigger.dev są dziś standardem: oba eliminują pułapki tradycyjnych kolejek (auto-retries, dead-letter). Na przykład Inngest automatycznie ponawia nieudane kroki [7], a Trigger.dev v3 gwarantuje brak timeoutów dzięki snapshotowaniu kontenera [4]. Vercel Cron przyda się do wywołań cyklicznych (np. „codziennie sprawdź logi”), ale zawsze tylko jako wyzwalacz – właściwa praca musi być delegowana do orchestratora.

2. Interfejs narzędziowy („Ręce”)

Wszystkie narzędzia (e-mail, GitHub, serwery, Jules AI itp.) warto udostępniać przez zunifikowany interfejs. Zalecane jest przyjęcie **MCP (Model Context Protocol)** – to otwarty standard (Anthropic 2024, wspierany przez Google/Microsoft) definiujący, jak LLM-y komunikują się z zewnętrznymi narzędziami i serwerami [8]. Dzięki MCP agent może *dynamicznie odkrywać* dostępne narzędzia – wystarczy, że dodamy kolejne endpointy (narzędzia) do serwera MCP, a agent automatycznie zobaczy je w czasie wykonywania [9] [10].

- **MCP jako standard:** Skonfiguruj serwer MCP (np. Gram lub wbudowany w Trigger.dev) i zarejestruj tam wszystkie funkcje narzędzi (nazwy, schematy JSON). Agent Lolek, używając klienta MCP, może zapytać o listę narzędzi i ich parametry [9]. Gdy w repo pojawi się nowa funkcja-narzędzie, wystarczy zaktualizować serwer MCP (w pipeline CI/CD), by agent sam „wykrył” nową usługę. Microsoft Agent Framework zwraca uwagę, że dzięki MCP agenci *dynamicznie odkrywają i wywołują zewnętrzne narzędzia* bez dodatkowego kodu klejącego [10].

- **Wzorce projektowe:** Każde narzędzie można też traktować jako moduł (plugin) w kodzie – np. umieścić je w katalogu `tools/` i ładować dynamicznie (np. poprzez `import()` i odczytanie metadanych). Alternatywnie użyć specyfikacji OpenAPI dla funkcji HTTP i automatycznie konwertować je na MCP (biblioteki jak Speakeasy czy trigger.dev MCP server pomagają w generowaniu opisów narzędzi). W każdym przypadku MCP zapewni, że Lolek dostanie aktualną listę „rąk" do dyspozycji.

- **Przykładowe biblioteki:** Next.js AI SDK (Vercel), Prisma ORM (Postgres), Inngest AgentKit lub Trigger.dev Tasks (zintegrowane z MCP), OpenAI/Anthropic SDK, GitHub Octokit, Nodemailer lub SendGrid (e-mail), inne API klientów (axios/fetch). Zastosuj także narzędzia do walidacji (np. zod) przy odbieraniu odpowiedzi z narzędzi.

3. Zdarzeniowy „loop świadomości"

Architektura musi reagować na przychodzące zdarzenia w cyklu: **„Zdarzenie -> Analiza AI -> Decyzja -> Akcja -> Zapis w pamięci"**. Technicznie można to zrealizować tak:

- **Endpoint** `/api/webhooks/incoming`**:** Tworzymy endpoint w Next.js (lub innym Frameworku) dedykowany przyjmowaniu webhooków (GitHub, systemy monitoringu, e-mail, Slack itp.). Ten endpoint **weryfikuje autentyczność** żądań (podpisy HMAC, tokeny API lub innych mechanizmów) i dopiero potem przetwarza dane. W celu bezpieczeństwa nadajemy listę dozwolonych źródeł lub klucze, by unikać zewnętrznych ataków.

- **Przekazanie do orkiestratora:** Zamiast blokować żądanie długą obróbką, po prostu **emitujemy zdarzenie** do systemu orkiestracji (np. Inngest lub Trigger.dev). Przykładowo wywołujemy `await inngest.send({ name: "external.event", data: eventData })` lub używamy SDK Triggera, co natychmiast rozpoczyna tło: zobacz przykład „Next.js webhook handler" uruchamiający zadanie w Trigger.dev [11]. Dzięki temu główny endpoint odsyła się szybko (HTTP 200) a praca idzie dalej w tle.

- **Analiza i decyzja:** W tle inna funkcja przeprowadza analizę – wywołuje LLM z kontekstem (treść e-maila, logi, stan serwerów, historia z bazy RAG). Model generuje plan działania lub decyzję. Wynikiem może być np. polecenie „wyślij maila", „otwórz ticket GitHub" czy „uruchom inny pod-agent".

- **Wykonanie akcji:** Na podstawie decyzji wyzwalamy kolejne zadania (narzędzia). Można skorzystać z MCP: agent wygeneruje wywołanie narzędzia (np. `tool: {name:"email_sender", args:{…}}`), które system przetłumaczy na rzeczywiste API/SDK. Każda akcja odbywa się jako osobny krok w workflow (czy to funkcja Node.js wysyłająca maila, czy wywołanie Jules AI), dzięki czemu cały proces jest asynchroniczny i odporny na błędy.

- **Zapisywanie do pamięci:** Po zakończeniu lub częściowym wykonaniu zadania trzeba **zalogować zdarzenie i wynik** do bazy pamięci (np. Postgres + wektorowy DB). Wzorzec inngestowy podpowiada: agent może wysłać specjalne zdarzenie „memory.add" do Inngest, a odizolowana funkcja zapisze dane do DB w tle [12]. Dzięki temu użytkownik nie czeka na zapisywanie, a system gwarantuje retry przy porażce (Inngest automatycznie powtórzy zapis [12]).

- **Bezpieczeństwo i walidacja:** Przy projektowaniu endpointa zadbajmy o sanitację danych (np. odfiltrowanie HTML, skryptów). Dobrą praktyką jest użycie nadzorczej warstwy (guardrails) na wejściu LLM, by ograniczyć niebezpieczne działania. Logujmy wszystkie zdarzenia i akcje (pod-obiekty Inngest czy Prometheus) dla pełnej przejrzystości.

Przykładowy przepływ danych

1. **Zdarzenie zewnętrzne** (GitHub webhook o nowym issue) trafia na `/api/webhooks/incoming`.
2. Endpoint weryfikuje podpis, normalizuje dane i wysyła `inngest.send({name:"github.issue.created", data:{title,body}})`.
3. Inngest uruchamia workflow: pierwszy krok to **LLM analizuje** treść issue + historię projektu.
4. Na podstawie odpowiedzi LLM, następny krok wyzwala narzędzie – np. `createGitHubComment(issueId, message)`.
5. Na koniec: wysyłane jest wydarzenie `memory.write` z podsumowaniem zdarzenia, a osobna funkcja zapisuje je do bazy (z retry) [12].

Stack technologiczny: Next.js (Vercel AI SDK) w TypeScript, Node.js, Prisma z PostgreSQL (i opcjonalnie wektorowy magazyn pamięci), Inngest/Trigger.dev (do workflow i MCP), biblioteki LLM (OpenAI lub Anthropic SDK), GitHub Octokit, nodemailer/SendGrid, a także narzędzia do kolejkowania (np. Upstash QStash) czy walidacji (zod). Całość działa serverless na Vercelu, z *de facto* brakiem własnej infrastruktury kolejkującej – wszystkie „kolejki" obsługuje Inngest/Trigger.

Bibliografia: Prezentowane rozwiązania opierają się na najnowszych praktykach 2025, m.in. Inngest (wieloetapowe funkcje zdarzeniowe) [2] [3], Trigger.dev v3 z CRIU (brak limitów) [4] oraz natywnych Cron jobach Vercel [6]. Standard MCP został przyjęty przez Anthropic/Google jako otwarty protokół narzędziowy [8] [10] i warto go zastosować do elastycznej integracji narzędzi. Z opisu przypadków (case studies) wiadomo, że dzielenie pracy na wiele niezależnych funkcji i wykorzystanie kolejek (Upstash QStash) efektywnie omija limity serwerless [1] [5].

---

1 5 Case Study: Solving Vercel's 10-Second Limit with QStash | by Kolby Sisk | Medium
https://medium.com/@kolbysisk/case-study-solving-vercels-10-second-limit-with-qstash-2bceeb35d29b

2 3 7 Background tasks, without the queues or workers - Inngest
https://www.inngest.com/uses/serverless-node-background-jobs

4 Trigger.dev v3: Durable Serverless functions. No timeouts. | Trigger.dev
https://trigger.dev/blog/v3-announcement

6 Cron Jobs
https://vercel.com/docs/cron-jobs

8 What is Model Context Protocol (MCP)? A guide | Google Cloud
https://cloud.google.com/discover/what-is-model-context-protocol

9 Extend your agent with Model Context Protocol - Microsoft Copilot Studio | Microsoft Learn
https://learn.microsoft.com/en-us/microsoft-copilot-studio/agent-extend-action-mcp

10 Introducing Microsoft Agent Framework: The Open-Source Engine for Agentic AI Apps | Microsoft Foundry Blog
https://devblogs.microsoft.com/foundry/introducing-microsoft-agent-framework-the-open-source-engine-for-agentic-ai-apps/

11 Example projects repo | Trigger.dev
https://trigger.dev/changelog/example-projects

12 Empowering Agents with Memory - Inngest Blog
https://www.inngest.com/blog/agent-memory-mem0