Architektura Bezpiecznego Wykonywania Kodu dla Autonomicznych Agentów AI: Raport Strategiczny 2025

Streszczenie Wykonawcze

W 2025 roku rozwój Dużych Modeli Językowych (LLM) osiągnął punkt zwrotny, przekształcając pasywne interfejsy konwersacyjne w aktywne, autonomiczne podmioty programistyczne. Projektowanie agenta takiego jak "Lolek", który posiada zdolność nie tylko do generowania syntaktycznie poprawnego kodu, ale także do jego uruchamiania, testowania i modyfikowania w czasie rzeczywistym, stawia przed architektami systemów bezprecedensowe wyzwania w zakresie bezpieczeństwa. Głównym problemem przestaje być jakość generowanego kodu, a staje się szczelność środowiska, w którym ten kod jest wykonywany. Umożliwienie modelowi probabilistycznemu, podatnemu na halucynacje i ataki typu *prompt injection*, bezpośredniego dostępu do zasobów obliczeniowych i systemów plików, tworzy wektor ataku o potencjalnie katastrofalnym promieniu rażenia.

Niniejszy raport stanowi wyczerpującą analizę technologiczną i architektoniczną krajobrazu "Code Execution Sandboxes" (piaskownic do wykonywania kodu) według stanu wiedzy na rok 2025. Opracowanie koncentruje się na trzech dominujących paradygmatach izolacji: zarządzanych platformach natywnych dla AI (reprezentowanych przez E2B), niskopoziomowej wirtualizacji (Firecracker) oraz izolacji na poziomie środowiska uruchomieniowego (WebAssembly). Analiza wykazuje, że o ile Firecracker pozostaje "złotym standardem" w zakresie izolacji infrastrukturalnej, to E2B wyrasta na lidera w zastosowaniach aplikacyjnych dzięki specjalizacji w obsłudze stanowych sesji AI i integracji z nowoczesnymi frameworkami. Z kolei WebAssembly, mimo obietnic związanych ze standardem WASI 0.2, wciąż boryka się z ograniczeniami w obsłudze pełnego stosu języka Python, co dyskwalifikuje go w wielu zaawansowanych scenariuszach *Data Science*.

Druga część raportu definiuje referencyjną architekturę integracji z Vercel AI SDK, kładąc nacisk na wzorce "Human-in-the-Loop" (HITL) jako niezbędny mechanizm kontrolny w systemach produkcyjnych. Ostatnia sekcja poświęcona jest bezpieczeństwu integracji z API GitHub, dekonstruując ryzyka związane z klasycznymi tokenami dostępu (PAT) i promując

model autoryzacji oparty na Aplikacjach GitHub (GitHub Apps) z precyzyjnie granulowanymi uprawnieniami, aby zapobiec atakom na łańcuch dostaw oprogramowania (Supply Chain Attacks).

---

1. Imperatyw Izolacji w Przepływach Pracy Agentów Autonomicznych

Ewolucja systemów AI od chatbotów do agentów sprawczych wymaga fundamentalnej zmiany w podejściu do bezpieczeństwa infrastruktury. W tradycyjnym modelu aplikacja webowa przetwarzała zaufany kod napisany przez inżynierów. W modelu agentowym, takim jak projektowany "Lolek", aplikacja wykonuje niezaufany, dynamicznie generowany kod, który z definicji należy traktować jako wrogi.

1.1 Profil Ryzyka Wykonywania Niezaufanego Kodu

Uruchamianie kodu generowanego przez LLM wiąże się z trzema kategoriami ryzyk krytycznych, które determinują wybór technologii izolacji.

Pierwszą i najbardziej oczywistą kategorią jest **złośliwa iniekcja kodu (Malicious Code Injection)**. Modele językowe są podatne na ataki, w których zmanipulowane wejście użytkownika (np. treść analizowanego pliku lub komentarz w kodzie) skłania model do wygenerowania instrukcji destrukcyjnych. Bez odpowiedniej izolacji, agent mógłby zostać zmuszony do skanowania sieci wewnętrznej, eksfiltracji zmiennych środowiskowych zawierających klucze API, czy instalacji oprogramowania typu *cryptominer*.[1] W 2025 roku obserwujemy wzrost wyrafinowania tych ataków, w tym techniki steganograficzne ukrywające instrukcje w obrazach lub plikach audio, które agent następnie przetwarza.

Drugą kategorią jest **wyczerpanie zasobów (Resource Exhaustion)**. Nawet bez złych intencji, halucynujący model może wygenerować nieskończone pętle, alokować gigabajty pamięci w rekurencyjnych funkcjach, czy doprowadzić do wycieku deskryptorów plików (tzw. fork bombs). W środowisku współdzielonym, takim jak tradycyjny kontener Docker, mogłoby to doprowadzić do ataku typu Denial of Service (DoS) na cały węzeł obliczeniowy, paraliżując działanie innych agentów lub usług systemowych.[1]

Trzecim ryzykiem jest **niezamierzona korupcja danych i stanu**. Agent, któremu powierzono zadanie operacji na plikach ("Lolek" ma pisać i testować kod), musi operować w przestrzeni,

gdzie błąd w logice (np. rm -rf na niewłaściwej ścieżce) nie spowoduje trwałej utraty danych konfiguracyjnych czy kodu źródłowego samej platformy. Wymaga to nie tylko izolacji procesora i pamięci, ale także ścisłej wirtualizacji systemu plików. Tradycyjne mechanizmy, takie jak chroot, są w 2025 roku uznawane za niewystarczające ze względu na liczne techniki ucieczki (escape techniques).[2]

1.2 Rola Stanowości w Konwersacjach Agentowych

Kluczowym aspektem, który odróżnia agentów programistycznych od typowych funkcji serverless, jest potrzeba zachowania stanu (statefulness). Gdy "Lolek" w pierwszej turze konwersacji tworzy plik main.py, a w drugiej turze użytkownik prosi o jego uruchomienie, system musi zagwarantować, że obie te operacje odbywają się w tym samym, trwałym kontekście.

Jeśli piaskownica jest restartowana po każdym wywołaniu LLM (model bezstanowy), agent traci swoją "pamięć operacyjną" – pliki, zainstalowane biblioteki, uruchomione procesy w tle. Wymuszałoby to na agencie ponowne generowanie i przesyłanie wszystkich plików przy każdej interakcji, co jest nieefektywne kosztowo i czasowo. Dlatego architektura piaskownicy dla agenta programistycznego musi wspierać długo żyjące sesje (long-running sessions), które mogą trwać od kilku minut do nawet 24 godzin, w przeciwieństwie do milisekundowych czasów życia funkcji Lambda.[3] To wymaganie dyskwalifikuje wiele prostych rozwiązań opartych na standardowych kontenerach uruchamianych w trybie *one-off*.

---

2. Analiza Porównawcza Technologii Piaskownic (Stan na 2025)

Rynek technologii izolacji dla AI uległ krystalizacji, wyłaniając trzy główne ścieżki rozwoju. Każda z nich oferuje inny balans między bezpieczeństwem, wydajnością a łatwością implementacji. Poniższa analiza dekonstruuje E2B, Firecracker oraz WebAssembly pod kątem przydatności dla autonomicznego agenta programistycznego.

2.1 E2B: Zarządzane Środowisko Natywne dla AI (AI-Native Managed Runtime)

E2B (Element to Byte) w 2025 roku ugruntowało swoją pozycję jako domyślny wybór dla zespołów budujących agentów, którzy nie chcą zarządzać własną infrastrukturą chmurową. Jest to rozwiązanie klasy *Platform-as-a-Service*, które abstrahuje złożoność wirtualizacji.[1]

Architektura i Możliwości

Fundamentem E2B jest orkiestracja mikro-maszyn wirtualnych (microVMs), najprawdopodobniej oparta na Firecrackerze, ale wzbogacona o warstwę zarządzania sesjami specyficzną dla LLM.5
Najważniejszą cechą E2B jest natywna obsługa stanowych sesji. Deweloper, korzystając z SDK (dostępnego dla Pythona i TypeScript), inicjuje sesję, która pozostaje aktywna ("alive") w chmurze E2B. Pozwala to agentowi na wykonywanie sekwencyjnych operacji: instalację pakietów (pip install), tworzenie struktury katalogów, a następnie uruchamianie testów jednostkowych w tym samym środowisku.3
Kolejnym wyróżnikiem jest **szybkość "zimnego startu" (cold start)**. Dzięki utrzymywaniu puli rozgrzanych instancji i optymalizacji obrazów systemowych, E2B jest w stanie dostarczyć gotową do pracy piaskownicę w czasie poniżej 200 milisekund.[1] Dla użytkownika końcowego oznacza to brak irytujących opóźnień podczas oczekiwania na wykonanie kodu przez agenta.

E2B rozwiązuje również problem **komunikacji dwukierunkowej**. W tradycyjnym podejściu przesłanie pliku do zdalnej maszyny wirtualnej i odebranie wygenerowanego wykresu wymagałoby budowy własnego API do transferu plików. E2B oferuje to "pudełkowo" – SDK automatycznie serializuje stdout, stderr, a także pliki (np. obrazy PNG wygenerowane przez matplotlib) i przesyła je z powrotem do aplikacji nadrzędnej.[6] Jest to kluczowe dla agentów *Data Science*, którzy muszą prezentować wyniki wizualne.

Model Ekonomiczny i Ograniczenia

Jako usługa zarządzana, E2B generuje koszty operacyjne, które skalują się liniowo wraz z czasem trwania sesji piaskownicy. W modelu "Hobby" oferuje darmowy pakiet kredytów, ale w zastosowaniach produkcyjnych (Pro) naliczane są opłaty za każdą sekundę aktywności wirtualnego CPU i GB pamięci RAM.[7] Dla projektu "Lolek", jeśli przewidywane jest masowe, ciągłe testowanie tysięcy przebiegów kodu dziennie, koszt chmury E2B może przewyższyć koszt własnej infrastruktury bare-metal. Dodatkowo, poleganie na zewnętrznym dostawcy wprowadza ryzyko *vendor lock-in* (uzależnienia od dostawcy), chociaż E2B udostępnia część

swojego kodu jako open-source, co teoretycznie umożliwia self-hosting.[4]

2.2 Firecracker: Fundament Infrastrukturalny

Firecracker, technologia rozwijana przez AWS i udostępniona jako open-source, stanowi bazę dla usług takich jak AWS Lambda czy Fargate. Jest to Monitor Maszyn Wirtualnych (VMM) wykorzystujący linuksowy KVM (Kernel-based Virtual Machine) do tworzenia mikroVM.

Architektura Bezpieczeństwa

Firecracker reprezentuje podejście "Hard Isolation". W przeciwieństwie do kontenerów Docker, które współdzielą jądro systemu operacyjnego hosta (korzystając z mechanizmów *cgroups* i *namespaces* do separacji), każda mikroVM Firecrackera posiada własne, zminimalizowane jądro gościa.[2] To drastycznie redukuje powierzchnię ataku. Nawet jeśli złośliwy kod zdoła wykorzystać błąd w jądrze gościa ("kernel panic" lub eskalacja uprawnień wewnątrz VM), nie daje to dostępu do hosta, ponieważ bariera wirtualizacji sprzętowej pozostaje nienaruszona.

Firecracker został zaprojektowany z myślą o minimalizmie. Nie emuluje pełnego zestawu sprzętu PC (jak QEMU), lecz jedynie niezbędne minimum: interfejsy sieciowe, dyskowe (virtio) i prosty kontroler klawiatury.[2] Dzięki temu narzut pamięciowy na jedną maszynę jest znikomy (<5 MB), co pozwala na upakowanie tysięcy mikroVM na jednym serwerze fizycznym ("High Density").[1]

Wyzwania Implementacyjne

Mimo niezaprzeczalnych zalet w zakresie bezpieczeństwa, Firecracker jest trudny w bezpośrednim użyciu dla twórców aplikacji.
Po pierwsze, wymaga dostępu do instrukcji wirtualizacji procesora (VT-x/AMD-V). Oznacza to, że uruchomienie go na typowej instancji w chmurze (np. AWS EC2) wymaga tzw. zagnieżdżonej wirtualizacji (nested virtualization), która bywa kosztowna i znacząco obniża wydajność.5 Idealnym środowiskiem dla Firecrackera są serwery typu bare metal.
Po drugie, Firecracker to tylko VMM. Nie posiada wbudowanego zarządzania obrazami dysków, sieciowania (CNI), ani API do przesyłania plików. Zbudowanie funkcjonalności "Code

Interpreter" na czystym Firecrackerze wymaga napisania własnej warstwy orkiestracji (control plane), która będzie zarządzać cyklem życia maszyn, przydzielać adresy IP i proxy'ować zapytania HTTP do wnętrza maszyn. Jest to zadanie inżynieryjne o dużej skali, opłacalne jedynie dla dużych organizacji budujących własną chmurę obliczeniową.3

2.3 WebAssembly (Wasm) i WASI: Lekka Alternatywa i Jej Granice

WebAssembly to binarny format instrukcji zaprojektowany dla wirtualnej maszyny opartej na stosie. Choć wywodzi się z przeglądarek, dzięki interfejsowi WASI (WebAssembly System Interface) zyskuje popularność po stronie serwera.

Mechanizm Izolacji

Wasm stosuje izolację programową (Software Fault Isolation - SFI). Kod wykonywany jest w wirtualnej maszynie, która matematycznie gwarantuje bezpieczeństwo pamięci. Moduł Wasm ma dostęp tylko do liniowego obszaru pamięci, który został mu przydzielony. Nie może "wyskoczyć" poza ten obszar ani odwołać się do zasobów systemowych (pliki, gniazda sieciowe), chyba że host (runtime, np. Wasmtime) jawnie mu na to zezwoli poprzez mechanizm capabilities.2
Zaletą tego podejścia jest ekstremalna szybkość. Uruchomienie modułu Wasm trwa mikrosekundy, ponieważ nie ma potrzeby bootowania systemu operacyjnego.10

Problem Ekosystemu Python w 2025

Dla agenta takiego jak "Lolek", który ma operować na kodzie Python, Wasm w 2025 roku wciąż przedstawia istotne ograniczenia. Standardowa implementacja CPython nie kompiluje się bezpośrednio do Wasm w sposób umożliwiający pełną funkcjonalność. Chociaż istnieją projekty takie jak Pyodide czy porty CPython do WASI, napotykają one na bariery:

1. **Brak obsługi gniazd sieciowych (Sockets):** Standard WASI w wersji *preview1* nie wspierał gniazd. Wersja *preview2* (WASI 0.2) wprowadziła pewne mechanizmy, ale obsługa pełnego stosu sieciowego w Pythonie (np. requests, urllib) wciąż wymaga skomplikowanych nakładek lub jest niemożliwa w czystym Wasm bez specjalnego hosta.[9]
2. **Brak wielowątkowości:** Model wątków w Wasm wciąż ewoluuje. Biblioteki Pythona polegające na wątkach OS (np. threading) mogą nie działać poprawnie lub działać w

sposób nieefektywny.[9]

3. **Wydajność bibliotek C-Extension:** Biblioteki takie jak numpy, pandas czy scipy, które są kluczowe dla analizy danych, w środowisku Linuksowym korzystają z wysoko optymalizowanego kodu C/Fortran. Ich portowanie do Wasm (np. w Pyodide) wiąże się ze spadkiem wydajności i problemami z kompatybilnością binarną. W wielu przypadkach uruchomienie pip install dowolna_biblioteka w środowisku Wasm po prostu się nie uda, jeśli biblioteka wymaga kompilacji kodu natywnego, dla którego nie ma gotowych binariów Wasm.[11]

Dlatego, mimo że Wasm jest obiecujący dla lekkich funkcji brzegowych (Edge Functions), dla ogólnego zastosowania "General Purpose Coding Agent" w 2025 roku jest technologią zbyt restrykcyjną w porównaniu do pełnego wirtualizowanego Linuksa oferowanego przez E2B czy Firecracker.

2.4 Tabela Porównawcza Technologii

Cecha	E2B	Firecracker	WebAssembly (Wasm)
Główny przypadek użycia	Agenci AI, Code Interpreters	Infrastruktura chmurowa, Multi-tenancy	Edge Computing, Wtyczki, Przeglądarka
Mechanizm Izolacji	MikroVM (bazuje na Firecracker)	Sprzętowa (KVM)	Programowa (SFI, Memory Safety)
Czas Startu	~200ms (bardzo szybki)	~125ms (ekstremalnie szybki)	~0.1ms (natychmiastowy)
Zarządzanie Stanem	**Natywne Sesje Stanowe**	Ulotne (wymaga orkiestracji)	Bezstanowe (zazwyczaj)
Wsparcie dla Python	**Pełne (Natywny Linux)**	**Pełne (Natywny Linux)**	Ograniczone (Pyodide, brak socketów)

Wsparcie sieciowe	Zarządzane, kontrolowany Egress	Pełne (konfigurowalne ręcznie)	Ograniczone (WASI capabilities)
Złożoność Integracji	Niska (Gotowe SDK)	Wysoka (Wymaga DevOps)	Średnia (Wymaga kompilacji)
Koszt	Model SaaS (za zużycie)	Tani przy dużej skali (Self-hosted)	Tani (małe zużycie zasobów)

Rekomendacja dla "Lolek": Biorąc pod uwagę wymóg operacji na plikach, instalacji pakietów i szerokiej kompatybilności z ekosystemem Python, **E2B jest jedynym logicznym wyborem** w 2025 roku dla zespołu, który chce skupić się na rozwoju inteligencji agenta, a nie na utrzymaniu infrastruktury wirtualizacyjnej. Firecracker jest alternatywą tylko w przypadku rygorystycznych wymogów *on-premise* i posiadania dedykowanego zespołu platformowego.

---

3. Orkiestracja i Integracja z Vercel AI SDK

Wybór technologii piaskownicy to dopiero połowa sukcesu. Drugą połową jest skuteczna integracja tego środowiska z "mózgiem" agenta, czyli modelem językowym, oraz interfejsem użytkownika. W ekosystemie JavaScript/TypeScript standardem w 2025 roku jest **Vercel AI SDK**.

3.1 Wzorce Architektoniczne "Tool Calling"

Fundamentem nowoczesnych agentów jest mechanizm "Tool Calling" (wywoływania narzędzi). LLM nie wykonuje kodu bezpośrednio; zamiast tego generuje ustrukturyzowane żądanie (zazwyczaj JSON) uruchomienia określonej funkcji. Vercel AI SDK (w szczególności biblioteka ai i moduł @ai-sdk/core) standaryzuje ten proces, abstrahując różnice między dostawcami modeli (OpenAI, Anthropic, Mistral).[12]

Dla agenta "Lolek" integracja E2B jako narzędzia w Vercel AI SDK wymaga zdefiniowania schematu wejściowego (np. za pomocą biblioteki zod) oraz funkcji wykonawczej.

Definicja Narzędzia (Kod Referencyjny)

Poniższy przykład ilustruje, jak zdefiniować narzędzie codeInterpreter, które łączy się z chmurą E2B, tworzy sesję i zwraca wyniki. Kluczowe jest użycie schematu zod do walidacji kodu generowanego przez AI.

TypeScript

```
import { z } from 'zod';
import { tool } from 'ai';
import { Sandbox } from '@e2b/code-interpreter';

export const codeInterpreterTool = tool({
 description: 'Wykonywanie kodu Python w bezpiecznej piaskownicy. Użyj do obliczeń, analizy danych i operacji na plikach.',
 inputSchema: z.object({
  code: z.string().describe('Kod Python do wykonania.'),
 }),
 execute: async ({ code }) => {
  // 1. Inicjalizacja piaskownicy (w produkcji warto reużywać ID sesji)
  const sandbox = await Sandbox.create();

  try {
   // 2. Wykonanie kodu w chmurze E2B
   const execution = await sandbox.runCode(code);

   // 3. Przechwycenie wyników (stdout, stderr, artefakty wizualne)
   return {
    stdout: execution.logs.stdout.join('\n'),
```

```
      stderr: execution.logs.stderr.join('\n'),
      // E2B zwraca np. obrazy jako base64, co jest kluczowe dla Data Science
      results: execution.results,
    };
  } catch (error) {
    return { error: `Błąd wykonania: ${error.message}` };
  } finally {
    // Ważne: Zarządzanie cyklem życia. W sesjach ciągłych nie zabijamy sandboxa natychmiast.
    // await sandbox.kill();
  }
 },
});
```

[13]

3.2 Pętla Agentowa i Samokorekta (Self-Correction Loop)

Autonomiczność agenta wynika z jego zdolności do samokorekty. Jeśli "Lolek" napisze kod, który zawiera błąd składniowy, piaskownica zwróci stderr z komunikatem błędu. Agent musi otrzymać ten komunikat z powrotem, przeanalizować go i wygenerować poprawioną wersję kodu.

W Vercel AI SDK realizuje się to poprzez parametr maxSteps w funkcji streamText (lub nowej abstrakcji ToolLoopAgent w wersji SDK 6 beta).[14] Ustawienie maxSteps: 5 pozwala na pięciokrotne przejście pętli: *Model -> Narzędzie -> Wynik -> Model*. Jest to krytyczne dla niezawodności. Badania wykazują, że dostarczenie informacji zwrotnej z stderr do modelu znacząco podnosi wskaźnik sukcesu w zadaniach programistycznych, umożliwiając agentowi "debugowanie samego siebie".[5]

Architektura ta wygląda następująco:

1. **Użytkownik:** "Oblicz 100-ny wyraz ciągu Fibonacciego."
2. **LLM:** Generuje kod Python (Tool Call).
3. **SDK:** Wykonuje kod w E2B.

4. **E2B:** Zwraca błąd (np. RecursionError).
5. **SDK:** Przekazuje błąd do LLM jako tool-result.
6. **LLM:** Analizuje błąd i generuje wersję iteracyjną (nie rekurencyjną).
7. **SDK:** Wykonuje poprawiony kod.
8. **E2B:** Zwraca poprawny wynik.
9. **LLM:** Formułuje odpowiedź końcową dla użytkownika.

3.3 Architektura "Human-in-the-Loop" (HITL) – Niezbędny Bezpiecznik

Dla agenta mającego wpływ na świat rzeczywisty (np. możliwość modyfikacji plików), pełna autonomia jest ryzykowna. Wzorce projektowe w 2025 roku kładą nacisk na architekturę "Human-in-the-Loop", gdzie człowiek pełni rolę ostatecznego decydenta przed wykonaniem akcji o wysokim ryzyku.

Implementacja HITL w Vercel AI SDK wymaga specyficznego podejścia: rozdzielenia *definicji* narzędzia od jego *wykonania*.[16]

Wzorzec implementacyjny:

1. **Server-Side:** W definicji narzędzia w api/chat/route.ts usuwamy funkcję execute lub stosujemy mechanizm activeTools (w SDK v6), aby zatrzymać automatyczne wykonanie.
2. **Klient (Frontend):** Używamy hooka useChat. Nasłuchujemy na wiadomości typu tool-call. Gdy taka wiadomość nadejdzie, zamiast automatycznie oczekiwać na odpowiedź serwera, renderujemy komponent UI (np. przyciski "Zatwierdź" / "Odrzuć") prezentujący proponowany kod.
3. **Interakcja:** Użytkownik weryfikuje kod.
    - Kliknięcie "Zatwierdź" wysyła sygnał do serwera (np. poprzez addToolOutput lub dedykowaną mutację), który dopiero wtedy uruchamia logikę E2B.
    - Kliknięcie "Odrzuć" wysyła do modelu informację o odmowie, co pozwala modelowi zaproponować alternatywę lub zaniechać działania.

Taka separacja odpowiedzialności (Model proponuje, Człowiek decyduje, System wykonuje) jest kluczowa dla budowania zaufania do autonomicznych systemów AI.[17]

---

4. Tożsamość i Autoryzacja: Bezpieczeństwo API

GitHub

Agent programistyczny bez dostępu do repozytorium kodu jest bezużyteczny. Jednak integracja z API GitHub stanowi jeden z najbardziej krytycznych wektorów ataku. Skompromitowanie poświadczeń agenta może doprowadzić do wstrzyknięcia złośliwego kodu do wszystkich projektów organizacji (atak typu Supply Chain).

4.1 Zmierzch Tokenów Osobistych (PAT)

W przeszłości powszechną praktyką było generowanie Personal Access Token (PAT) dla konta użytkownika i przekazywanie go botowi. W 2025 roku jest to uznawane za **antywzorzec bezpieczeństwa**.[18]

- **Brak rozliczalności:** Działania agenta są logowane jako działania użytkownika ("committed by Jan Kowalski"), co uniemożliwia audyt i odróżnienie zmian wprowadzonych przez człowieka od tych wprowadzonych przez AI.
- **Zbyt szeroki zakres:** Klasyczne PAT-y często mają uprawnienia repo (pełny dostęp do wszystkich repozytoriów prywatnych i publicznych), co narusza zasadę najmniejszych uprawnień (PoLP).
- **Problemy z cyklem życia:** PAT-y są długowieczne. Ich rotacja jest ręczna i często zaniedbywana. Wyciek PAT oznacza trwały dostęp dla atakującego do czasu jego ręcznego unieważnienia.

4.2 Standard Przemysłowy: GitHub Apps

Dla autonomicznych agentów takich jak "Lolek", jedynym akceptowalnym standardem jest autoryzacja poprzez **GitHub App**.[19]

Mechanizm Działania

Aplikacja GitHub App posiada własną tożsamość ("Lolek [bot]"). Nie loguje się hasłem, lecz wykorzystuje parę kluczy kryptograficznych (klucz prywatny przechowywany bezpiecznie po

stronie serwera agenta, klucz publiczny w GitHub).
Proces uwierzytelniania przebiega następująco:

1. Agent generuje token JWT (JSON Web Token) podpisany swoim kluczem prywatnym.
2. JWT jest wymieniany w API GitHub na **Installation Access Token**.
3. Token ten jest ważny tylko przez 1 godzinę i ma zakres uprawnień ściśle ograniczony do konkretnej instalacji (np. tylko do jednego repozytorium, w którym agent został włączony).[20]

Granulacja Uprawnień (Fine-Grained Permissions)

GitHub Apps pozwalają na niezwykle precyzyjne definiowanie uprawnień. Dla agenta programistycznego kluczowe jest rozróżnienie między dostępem do kodu a dostępem do infrastruktury CI/CD.

Krytyczna rekomendacja bezpieczeństwa: Należy bezwzględnie zablokować uprawnienie workflows:write.21
Dlaczego? Jeśli agent ma prawo modyfikować pliki w katalogu .github/workflows, może zmienić definicję procesu budowania aplikacji, dodając np. krok przesyłający wszystkie sekrety środowiskowe (klucze AWS, hasła do bazy danych) na zewnętrzny serwer. Jest to klasyczny scenariusz ataku na łańcuch dostaw.
Poprawna konfiguracja manifestu aplikacji dla "Loleka" powinna zawierać:

- contents: read (ewentualnie write tylko jeśli agent commituje bezpośrednio, co jest niezalecane).
- pull_requests: write (aby agent mógł zgłaszać zmiany poprzez PR, co wymusza *Code Review*).
- issues: read (do pobierania kontekstu zadań).
- **Brak dostępu** do admin, secrets, workflows.[22]

4.3 Zapobieganie Eskalacji Uprawnień

Nawet przy użyciu GitHub Apps, należy stosować mechanizmy *defense-in-depth*:

1. **Branch Protection Rules:** Skonfigurowanie repozytorium tak, aby wymagało zatwierdzenia przez człowieka (Code Owner) przed scaleniem jakiegokolwiek PR-a od bota. Agent nigdy nie powinien mieć możliwości pushowania bezpośrednio do gałęzi main.[23]
2. **Secret Scanning:** Włączenie automatycznego skanowania commitów. Jeśli "Lolek" w

wyniku halucynacji wpisze w kodzie klucz API, GitHub zablokuje taki commit przed przyjęciem go przez serwer.[23]

3. **Izolacja sieciowa:** Jeśli to możliwe, agent powinien łączyć się z GitHubem z adresów IP znajdujących się na białej liście organizacji.

---

5. Podsumowanie i Rekomendacje Wdrożeniowe

Zaprojektowanie bezpiecznego agenta "Lolek" w 2025 roku wymaga odejścia od prostych skryptów na rzecz zaawansowanej architektury systemowej. Analiza wykazała, że:

1. **Technologia Piaskownicy: E2B** jest rekomendowanym rozwiązaniem. Zapewnia ono niezbędną równowagę między bezpieczeństwem (mikroVM) a użytecznością (stanowość, wsparcie dla Python/Data Science), której brakuje obecnie rozwiązaniom opartym na czystym Wasm czy surowym Firecrackerze.
2. **Warstwa Orkiestracji:** Wykorzystanie **Vercel AI SDK** z implementacją wzorca **Human-in-the-Loop** jest krytyczne. Nie wolno pozwalać agentowi na "wolną rękę" w operacjach zapisu; każda destrukcyjna akcja musi być poprzedzona jawną zgodą użytkownika w interfejsie.
3. **Bezpieczeństwo Tożsamości:** Należy całkowicie wyeliminować Tokeny Osobiste (PAT) na rzecz **GitHub Apps**. Konfiguracja uprawnień musi realizować zasadę najmniejszego przywileju, ze szczególnym naciskiem na ochronę plików konfiguracyjnych CI/CD (workflows).

Wdrożenie tych rekomendacji pozwoli na stworzenie asystenta programistycznego, który realnie przyspiesza pracę deweloperów, nie stając się jednocześnie koniem trojańskim w infrastrukturze organizacji. Przyszłość należy do agentów, którzy są nie tylko inteligentni, ale przede wszystkim przewidywalni i bezpieczni.

Works cited

1. Top AI Code Sandbox Products in 2025 | Modal Blog, accessed November 25, 2025, https://modal.com/blog/top-code-agent-sandbox-products
2. restyler/awesome-sandbox: Awesome Code Sandboxing for AI - GitHub, accessed November 25, 2025, https://github.com/restyler/awesome-sandbox
3. Top Vercel Sandbox alternatives for secure AI code execution and sandbox environments | Blog — Northflank, accessed November 25, 2025, https://northflank.com/blog/top-vercel-sandbox-alternatives-for-secure-ai-code-execution-and-sandbox-environments
4. Code Interpreter Sandbox — E2B Blog, accessed November 25, 2025, https://e2b.dev/blog/e2b-sandbox
5. Chris Hay's Code Sandbox MCP Server: A Deep Dive for AI Engineers, accessed

November 25, 2025, https://skywork.ai/skypage/en/chris-hays-code-sandbox-ai-engineers/1980120660590239744
6. Launching the Code Interpreter SDK — E2B Blog, accessed November 25, 2025, https://e2b.dev/blog/launching-the-code-interpreter-sdk
7. Pricing - E2B, accessed November 25, 2025, https://e2b.dev/pricing
8. Top Modal Sandboxes alternatives for secure AI code execution | Blog - Northflank, accessed November 25, 2025, https://northflank.com/blog/top-modal-sandboxes-alternatives-for-secure-ai-code-execution
9. WASI and the WebAssembly Component Model: Current Status - eunomia, accessed November 25, 2025, https://eunomia.dev/blog/2025/02/16/wasi-and-the-webassembly-component-model-current-status/
10. WebAssembly in Modern Development: Analysis 2024-2025 - DEV Community, accessed November 25, 2025, https://dev.to/hashbyt/webassembly-in-modern-development-analysis-2024-2025-3k2i
11. Experimental - Python for the Web - Visual Studio Marketplace, accessed November 25, 2025, https://marketplace.visualstudio.com/items?itemName=ms-vscode.vscode-python-web-wasm
12. AI SDK - Vercel, accessed November 25, 2025, https://vercel.com/docs/ai-sdk
13. E2B | The Enterprise AI Agent Cloud, accessed November 25, 2025, https://e2b.dev/
14. AI SDK 6 Beta, accessed November 25, 2025, https://ai-sdk.dev/docs/announcing-ai-sdk-6-beta
15. Can LLMs Correct Themselves? A Benchmark of Self-Correction in LLMs - arXiv, accessed November 25, 2025, https://arxiv.org/html/2510.16062v2
16. Next.js: Human-in-the-Loop Agent with Next.js - AI SDK, accessed November 25, 2025, https://ai-sdk.dev/cookbook/next/human-in-the-loop
17. Lead Agent - Vercel, accessed November 25, 2025, https://vercel.com/templates/next.js/lead-processing-agent
18. Replacing a GitHub Personal Access Token with a GitHub Application - Aembit, accessed November 25, 2025, https://aembit.io/blog/replacing-a-github-personal-access-token-with-a-github-application/
19. Registering a GitHub App from a manifest, accessed November 25, 2025, https://docs.github.com/en/apps/sharing-github-apps/registering-a-github-app-from-a-manifest
20. GitHub Agent HQ Security and Privacy: Protecting Your Code with AI Agents - Skywork.ai, accessed November 25, 2025, https://skywork.ai/blog/agent/github-agent-hq-security-and-privacy-protecting-your-code-with-ai-agents/
21. Permissions required for fine-grained personal access tokens - GitHub Docs,

accessed November 25, 2025, https://docs.github.com/en/rest/authentication/permissions-required-for-fine-grained-personal-access-tokens
22. How to Implement Fine-Grained Access Control in GitHub: A Step-by-Step Guide, accessed November 25, 2025, https://ones.com/blog/implement-fine-grained-access-control-github/
23. About GitHub Copilot coding agent, accessed November 25, 2025, https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-coding-agent