

פרויקט גמר: סוכן AI היברידי מונע אירועים עם Event Sourcing ו-Kafka Streams

1. מטרת הפרויקט

פרויקט גמר זה יעמיק את ההבנה והיישום של ארכיטקטורות מבוססות אירועים (Event-Driven Architectures) עבור סוכני AI מתקדמים. נהפוך את הבוט ל מכונת מצב אסינכרונית ומונעת אירועים לחלוטין, שבה Apache Kafka אינו רק "צינור" להעברת הודעות, אלא הופך להיות: מקור האמת היחיד (Single Source of Truth): כל שינוי במצב הסוכן (קבלת שאילתה, יצירת תוכנית, ביצוע כלי, קבלת תוצאה, ניסוח תשובה) נרשם כאירוע בלתי משתנה (Immutable Event) ב-Kafka. פלטפורמה לעיבוד זרמים עם מצב (Stateful Stream Processing): רכיבים קריטיים במערכת (כמו Aggregator ו-Orchestrator) יבנו ויעדכנו את המצב הפנימי שלהם (למשל, סטטוס Plan עבור conversationId ספציפי) ישירות מתוך זרמי האירועים של קפקא. זה מבטיח עמידות לכשלים, סקלאביליות ויכולת שחזור. מנגנון תזמור מבוזר (Distributed Orchestration Mechanism): במקום לוגיקה טורית או ריכוזית, תזמור הכלים מתבצע על ידי רצף אירועים, כאשר כל אירוע מפעיל את הצעד הבא ב-Pipeline. בפרויקט זה נתרגל:

- Event Sourcing: מידול מצב המערכת כזרם אירועים.
- Kafka Streams (או מימוש עקרונית): בניית רכיבים בעלי מצב (Stateful) המעבדים זרמי אירועים.
- CQRS (Command Query Responsibility Segregation): הפרדה בין פקודות (Requests) לאירועים (Events) ב-Kafka.
- Fault Tolerance & Resilience: המערכת תהיה עמידה לכשלים, תוך יכולת שחזור מצב אוטומטי מ-Kafka.
- Microservices Architecture: פיצול המערכת ל-Microservices עצמאיים המבצעים פונקציה אחת בלבד ומתקשרים באמצעות Kafka.

2. ארכיטקטורת המערכת - Event-Sourced AI State Machine

הזרימה הכללית:

- קלט מהמשתמש: UserCommandTopic (פקודה: UserQueryReceived).
- Router Agent: מפיק אירוע PlanGenerated ל-ConversationEventsTopic.
- Orchestrator Agent: בונה מצב של ה-Plan מ-ConversationEventsTopic ופולט פקודות ToolInvocationRequested ל-ToolCommandsTopic.
- Tool Workers: מאזינים ל-ToolCommandsTopic, מבצעים פעולה, ופולטים אירועי ToolInvocationResulted ל-ConversationEventsTopic.
- Orchestrator Agent (ממשיך): מעדכן את מצב ה-Plan, מפעיל צעדים נוספים, או מפיק אירוע PlanCompleted ל-ConversationEventsTopic.
- Aggregator Agent: מאזין ל-ConversationEventsTopic, מזהה PlanCompleted, ופולט פקודה SynthesizeFinalAnswerRequested ל-UserCommandTopic.
- Synthesis Worker: מאזין ל-UserCommandTopic, מנסה תשובה ופולט אירוע FinalAnswerSynthesized ל-ConversationEventsTopic.
- User Interface: מציג את התשובה הסופית למשתמש.

3. דרישות פונקציונליות וארכיטקטוניות

א. Kafka Core Infrastructure & Schema Registry

הקמת Kafka: הרצת Kafka ו-Zookeeper (מומלץ ב-Docker Compose).

Topics:

- user-commands: קלט מהמשתמש ופקודות פנימיות למערכת.
- conversation-events: ה-Event Log הראשי. כל אירועי שינוי מצב הקשורים לאינטראקציה של סוכן (PlanGenerated, ToolInvocationResulted, PlanStepCompleted, FinalAnswerSynthesized).
- tool-invocation-requests: בקשות (פקודות) ספציפיות לביצוע כלי.
- dead-letter-queue: לטיפול בהודעות שלא עובדו בהצלחה.
- Schema Registry (קונספטואלי/מימוש פשוט): הגדירו סכימות JSON/Avro לכל סוג אירוע ב-conversation-events ו-user-commands. קודמו חייבים לוודא שהם מצייתים לסכימות אלו. (לדוגמה, שמרו קבצי json המגדירים את הסכימה עבור כל אירוע).

ב. User Interface & Command Producer (Node.js)

user-interface.ts:

- קולט קלט מהמשתמש (CLI).
- יוצר conversationId ייחודי (UUID), ו-timestamp.
- בונה אובייקט UserQueryReceived (פקודה).
- Producer: שולח את הפקודה ל-Kafka Topic: user-commands.
- Consumer: מאזין ל-conversation-events לחיפוש אירוע FinalAnswerSynthesized עם ה-conversationId התואם, ומציג את התשובה למשתמש.

ג. Agent Router (Node.js - Consumer/Producer)

router.ts:

- Consumer Group: מאזין ל-user-commands (ספציפית לפקודות UserQueryReceived).
- LLM (Ollama/OpenAI Fallback): מפעיל את ה-Router (עם ה-ROUTER_SYSTEM_PROMPT המעודכן) כדי לזהות כוונות וליצור plan JSON מורכב.
- Producer: שולח אירוע PlanGenerated (כולל conversationId ו-plan JSON) ל-Kafka Topic: conversation-events.

ד. Stateful Plan Orchestrator (Node.js - Kafka Streams-like Logic)

זהו רכיב קריטי המנהל את המצב של כל plan באופן עמיד וסקלאבילי.

orchestrator.ts:

Consumer Group: מאזין לשיני Topics:

- conversation-events (לאירועי PlanGenerated ו-ToolInvocationResulted).
- tool-invocation-requests (כדי לוודא שהוא לא שולח בקשה כפולה).

State Store (Persistent Key-Value Store): בנה/שחזר את מצב ה-Plan עבור כל conversationId. ניתן להשתמש ב-RocksDB, LevelDB (דרך level-rocksdb ב-Node.js), או לנהל Map בזיכרון אשר מגובה על ידי היכולת לשחזר את המצב מה-conversation-events topic בעת הפעלה מחדש.

Key: conversationId

Value: אובייקט המכיל את ה-plan המלא, stepIndex נוכחי, תוצאות ביניים, וסטטוס (PENDING, RUNNING, COMPLETED, FAILED).

לוגיקת עיבוד אירועים:

- כאשר מגיע PlanGenerated: אתחל את מצב ה-Plan ב-State Store.
- כאשר מגיע ToolInvocationResulted: עדכן את מצב ה-Plan ב-State Store עם התוצאה.
- Producer (פקודות): על בסיס מצב ה-Plan המעודכן ב-State Store:
- קבל החלטה על הצעד הבא (כולל החלפת Placeholders).
- שלח פקודה ToolInvocationRequested ל-Kafka Topic: tool-invocation-requests.
- אם ה-Plan הושלם: שלח אירוע PlanCompleted ל-Kafka Topic: conversation-events.
- אם ה-Plan נכשל: שלח אירוע PlanFailed ל-Kafka Topic: conversation-events.

ה. Stateless Tool Workers (Python & Node.js)

כל כלי יהיה Consumer Group עצמאי, המבצע פונקציה אחת ופולט אירוע ToolInvocationResulted. הם צריכים להיות Idempotent.

(Python Consumer/Producer) rag-retriever-worker.py:

- Consumer: מאזין ל-tool-invocation-requests עבור tool: getProductInformation.
- Producer: שולח אירוע ToolInvocationResulted ל-Kafka Topic: conversation-events.
- (Node.js Consumer/Producer) llm-inference-worker.ts:
- Consumer: מאזין ל-tool-invocation-requests עבור tool: generalChat, ragGeneration, orchestrationSynthesis.
- Producer: שולח אירוע ToolInvocationResulted ל-Kafka Topic: conversation-events.
- math-worker.ts, weather-worker.ts, exchange-rate-worker.ts: (כנ"ל).
- כל worker שולח את התוצאה שלו כאירוע ToolInvocationResulted ל-conversation-events Topic.
- 1. (Node.js - Kafka Streams-like Logic) Aggregator & Final Synthesis

(Node.js Consumer/Producer) aggregator.ts:

- Consumer Group: מאזין ל-Kafka Topic: conversation-events (ספציפית לאירועי PlanCompleted).
- State Store (Optional): במקרה זה, הוא יכול לבנות את תוצאות ה-Plan על בסיס אירועי ה-ToolInvocationResulted הקודמים עבור ה-conversationId שנשלח ב-PlanCompleted.
- Producer: שולח פקודה SynthesizeFinalAnswerRequested (עם כל תוצאות הביניים) ל-Kafka Topic: user-commands.
- (Node.js Consumer/Producer) synthesis-worker.ts:
- Consumer Group: מאזין ל-Kafka Topic: user-commands (ספציפית לפקודות SynthesizeFinalAnswerRequested).
- מפעיל את ה-ORCHESTRATION_SYNTHESIS_PROMPT עם OpenAI.

- Producer: שולח אירוע FinalAnswerSynthesized ל-Kafka Topic: conversation-events.

4. ניתוח ביצועים, עמידות ובחירת מודלים (Benchmarking & Resilience)

מדידת זמנים (End-to-End Latency):

מעקב אירועים: כל אירוע (UserQueryReceived, PlanGenerated, ToolInvocationRequested, ToolInvocationResulted, FinalAnswerSynthesized) חייב להכיל conversationId ו-timestamp של יצירת האירוע. ה-user-interface או שירות נפרד יוכל לחשב את זמן ה-End-to-End Latency על ידי איסוף כל האירועים עבור conversationId נתון.

מדדים: End-to-End Latency, Latency per Worker, Throughput (events/sec), Consumer Lag.

תרחישי Resilience חובה להדגים:

- קריסת Worker: הריצו Plan מורכב. כבו את אחד מ-Python Workers באמצע. הציגו כי ה-Plan נתקע. הרימו את ה-Worker שוב. הציגו שה-Plan ממשיך מאיפה שהפסיק.
- קריסת Orchestrator: כבו את ה-orchestrator.ts באמצע. הרימו אותו מחדש. הציגו שהוא משחזר את מצבו מה-conversation-events וממשיך את Plan.
- Duplicate Events: הדגו את טיפול ה-Workers באירועים כפולים (לדוגמה, על ידי שיגור ידני של אותו ToolInvocationRequested פעמיים).

טבלת Benchmarking מורחבת (בדוח README.md):

רכיב המערכת / תרחיש	מודל (ספק)	זמן עיבוד ממוצע לאירוע (ms)	קצב אירועים מרבי (Events/sec)	איכות / דיוק (1-5)	עלות משוערת
Router (PlanGenerated)	Ollama (Llama3)	?	0
Router (Fallback)	OpenAI GPT-3.5	5	\$
Orchestrator (ToolInvocationRequested)	Stateful Processor	N/A	0
Tool: RAG Retrieval	HF Embedding (Python)	5	0
Tool: LLM Infer (Ollama)	Ollama (Llama3)	?	0
Tool: LLM Infer (OpenAI)	OpenAI GPT-3.5	5	\$
Aggregator (SynthesizeFinalAnswerRequested)	Stateful Processor	N/A	0
Final Synthesis	OpenAI GPT-3.5	5	\$
End-to-End Latency (Complex Plan)	Total over multiple events	5	Total

ניתוח ומסקנות (מורחב ב-README.md):

- הצדיקו את בחירת Kafka כ-Event Store: כיצד שימוש ב-Event Sourcing משפר את עמידות המערכת, יכולת השחזור ו-Auditability (מעקב אחר כל פעולה)?
 - Stateful Processing Power: הסבירו כיצד ה-orchestrator ו-aggregator ניצלו את זרמי האירועים של קפקא כדי לנהל מצב בצורה מבוצרת ועמידה, במקום להסתמך על זיכרון נדיף או DB חיצוני לכל שינוי.
 - יתרונות CQRS ו-Idempotency: כיצד הפרדת Command/Event והקפדה על Idempotency תורמים לאמינות במערכת מבוצרת ואסינכרונית?
 - Trade-offs של Event Sourcing: מהם האתגרים בפתרון זה (מורכבות, Eventual, debugging, Consistency)?
- שיפורים עתידיים: הצעות לשיפורים נוספים כמו Kafka Streams DSL (עבור Node.js יש ספריות כמו kafka-streams, Confluent Schema Registry, KSQL, Observability tools, Grafana/Prometheus).

5. תוצרי הגשה

- קוד מקור מלא: (מפוצל לקבצים שונים תחת src/node ו-src/python).
- מודל אירועים (Schemas): תיקיית src/schemas עם קבצי JSON/Avro לכל סוג אירוע ופקודה.
- כל רכיב (router.ts, orchestrator.ts, Workers) ימומש כ-Microservice עם Consumer Group משלו.
- /prompts: כל הפרומפטים מעודכנים ומופרדים.
- /data/products: קבצי מוצרים.
- docker-compose.yml: להרמת Kafka, Zookeeper, ChromaDB, Ollama, וכל המיקרו-שירותים (Node/Python workers).
- README.md: מפורט כנדרש, כולל:
- דיאגרמת ארכיטקטורה.
- הסבר מפורט על Event Sourcing ו-Stateful Stream Processing בקונטקסט של הפרויקט.
- הוראות הפעלה מלאות.
- דוחות Benchmarking, Resilience Analysis ומסקנות מפורטות.
- קובץ לוג לדוגמה (Execution Log):
- לפחות 3 תרחישי Orchestration מורכבים.
- לפחות 2 תרחישי RAG.
- חובה: הדגמה של תרחישי Resilience (קריסה ושחזור).

בהצלחה רבה בפרויקט!