

בניית Knowledge Graph בעזרת agents: קורס DeepLearning.AI ו-Neo4j

איך מתכננים ובונים knowledge graph מנתונים מובנים ולא מובנים עם מערכת multi-agent ב-Google ADK, ומה מכסים שלושת השיעורים הראשונים

· Andrew Ng, Andreas Kollegger · DeepLearning.AI × Neo4j (קורס קצר, הועלה מחדש ב-X על ידי Morty) ·
תאריך הריפוסט: 2026-10-01

משך: כ-66 דקות · X

סיכום אוטומטי — AI Briefing (aibriefing.dev)

על המסמך הזה

ההקלטה היא שלושת השיעורים הראשונים של הקורס הקצר Agentic Knowledge Graph Construction, ש-DeepLearning.AI הפיקה יחד עם Neo4j. Andrew Ng פותח בהקדמה קצרה, ואת רוב ההקלטה מעביר Andreas Kollegger, developer evangelist ל-GenAI ב-Neo4j. בתמלול השם יצא Andres Colliger, ובשקף הפתיחה הוא כתוב Andreas Kollegger. בפוסט ב-X נטען ש-Andrew Ng הוציא קורס של שעה על בניית מערכות agentic מאפס. ההקלטה לא תומכת בזה: Ng רק מציג את הקורס, הנושא הוא בניית knowledge graph, והפרק Why Graphs are the future שמופיע בתיאור ב-01:06:03 לא קיים בהקלטה.

הטענה המרכזית: knowledge graph שמחבר chunks של טקסט ל-entities ולנתונים מובנים נותן ל-LLM הקשר מדויק יותר מ-RAG רגיל. את הצעד הכי קשה, תכנון ה-schema, אפשר להעביר למערכת multi-agent. ה-agents לא בונים את הגרף בעצמם. הם מנהלים שיחה עם המשתמש ומפיקים תוכניות בנייה, ואת הבנייה עצמה מבצע tool דטרמיניסטי. בחלק השני של ההקלטה עוברים על Google ADK בפועל: agent בודד, runner, tools ולולאת events, ואחר כך צוות agents עם delegation ו-session state.

איך לקרוא את המסמך

- 1. **תקציר מנהלים:** הטענות המרכזיות עם timestamps, ומה לא נאמר בהקלטה.
- 2. **מהו knowledge graph:** הדרך מ-join tables לשאילתת Cypher של המלצות.
- 3. **מקרה הבוחן:** ניתוח root cause לתקלות מוצר, ושלושת תת-הגרפים domain, lexical ו-subject.
- 4. **agent ו-multi-agent:** ההגדרה ההנדסית של agent, היתרונות והחסרונות שלו, ודפוסי delegation.
- 5. **ארכיטקטורת המערכת:** ה-agents שבונים את הגרף, והתוכניות שהם מפיקים.
- 6. **Google ADK בפועל:** agent ראשון, delegation, runner, tools ו-state, כולל מה שהשתבש בדמו.
- 7. **מסקנות ופעולות:** מה כדאי לעשות עכשיו, וטבלת שאלות פתוחות.
- 8. **מילון מונחים:** המונחים באנגלית שמופיעים במסמך.

הערות מקור המסמך הופק אוטומטית מהתמלול ומהשקפים של ההקלטה. ייתכן שזיהוי הדיבור (ASR) שיבש שמות ומונחים. למשל, knowledge graph תומלל לפעמים knowledge draft, ו-ADK תומלל AVK. ציטוטים מובאים מילה במילה ונבדקו מול ה-timestamp שלהם. לא נוסף מידע ממקורות חיצוניים.

1. תקציר מנהלים

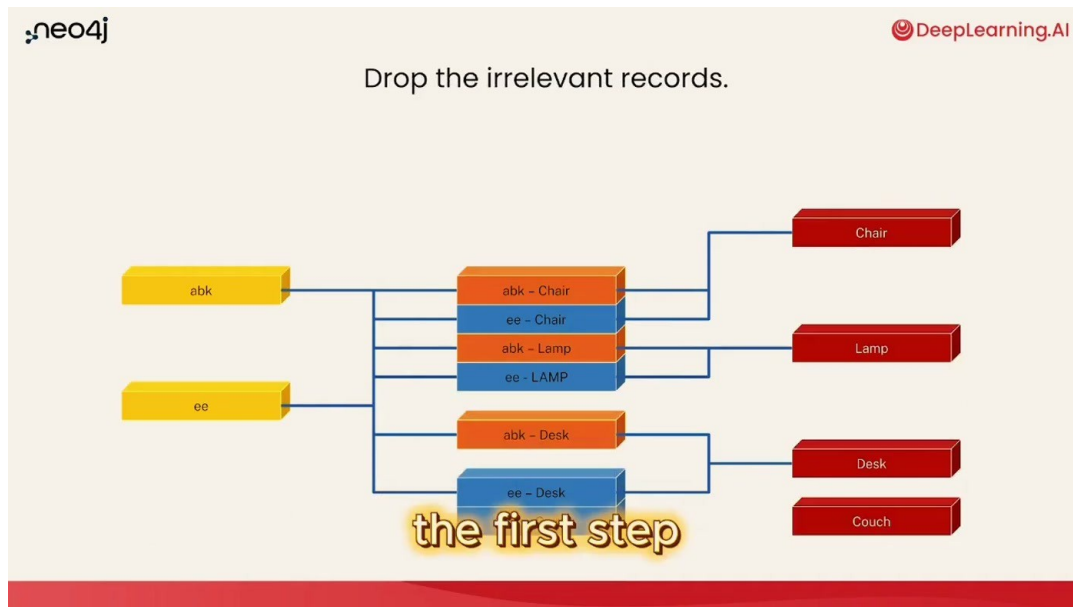
ההקלטה מכסה את שיעורים 1 עד 3 מתוך קורס של שמונה שיעורים. שיעור 1 מסביר מה זה knowledge graph ומציג את מקרה הבוחן. שיעור 2 מציג את ארכיטקטורת ה-agents. שיעור 3 הוא מבוא מעשי ל-Google ADK. שיעורים 4 עד 8, שבהם בונים בפועל את ה-agents של הגרף, מופיעים רק כמפת דרכים על שקף.

- **ב-knowledge graph הקשרים הם רשומות מלאות ולא join tables:** לכל relationship יש כיוון, type אחד ו-properties משלו. שאליות ב-Cypher נקראות כמעט כמו משפט באנגלית, ולכן מתאימות לעבודה עם LLMs [7:41].
- **הגרף מורכב משלושה תת-גרפים:** domain graph מקובצי CSV, lexical graph מ-chunks של markdown, ו-subject graph של entities שחולצו מהטקסט וממופים למוצרים בנתונים המובנים [12:23].
- **agent הוא סוג חדש של control flow:** לולאה, קריאה ל-LLM ו-switch שמריץ פעולות בקוד. החסרונות: איטי, לא דטרמיניסטי, ועלות ה-tokens מטפסת מהר בפרודקשן [15:26, 13:55].
- **ב-multi-agent יש שני מנגנוני אינטראקציה:** delegation בין agents, ו-agent שנקרא כ-tool. בכל מעבר כזה אפשר לשלב checkpoint מול המשתמש [15:26].
- **ה-agents מפקים תוכניות ולא גרף:** Graph Construction Plan ו-Knowledge Extraction Plan. את הבנייה מבצע tool יחיד שרץ בלולאה על כללי הבנייה [21:32, 20:01].
- **ה-schema נבנה בדפוס critic:** זוג agents, אחד מציע schema והשני מבקר, והם חוזרים על הסבב עד שמתקבל schema שמתאים לקבצים שאושרו ולמטרת המשתמש [20:01].
- **ב-ADK רוב העבודה היא ניסוח הוראות:** ה-description של agent קובע מתי מעבירים אליו עבודה, ה-docstring של tool קובע מתי קוראים לו, ושיפור ה-instructions הוא העבודה העיקרית [49:00, 29:10].
- **ב-delegation עובר כל הקשר השיחה וגם השליטה בה:** בדמו, ה-greeting sub-agent נשאר בשליטה והעביר בעצמו את השיחה ל-farewell agent, בלי לחזור קודם ל-root [50:35, 55:09].
- **session state הוא dictionary משותף לכל ה-agents, וה-LLM עדיין ממציא:** גישה ל-state דרך tool_context או output_key. בדמו האינטראקטיבי המודל המציא שם משתמש לפני שהמשתמש אמר אותו [1:04:22, 58:14].

מה לא נאמר

- **אין מדידה:** לא מוצגים נתוני דיוק, latency או עלות שמשווים knowledge graph ל-RAG רגיל. ההצהרה שהתשובות מדויקות יותר לא מגובה במספרים.
- **הבנייה עצמה לא מוצגת:** שיעורים 4 עד 8 (user intent, file suggestion, schema proposal, entity and fact types, ה-tool שבונה את הגרף) לא נכללים בהקלטה. יש רק שקף מפת דרכים.
- **חיבור ה-entities למוצרים נשאר מעורפל:** איך entity שחולץ מטקסט ממופה בפועל למוצר בנתונים המובנים (CORRESPONDS_TO) לא מוסבר מעבר לרמה העקרונית.
- **יש קטע חסר:** בתמלול חסר הקטע שבין 32:11 ל-35:17. שם כנראה הוסבר שדה ה-instruction של ה-agent והוצג ה-runner.
- **אין התייחסות לפרודקשן:** כל הדמו רץ על persistence. InMemorySessionService, ניהול כמה sessions ו-evaluation מוזכרים רק במשפט אחד.
- **הטענות בפוסט ב-X לא מבוססות:** ההקלטה לא תומכת בשום דבר על כך שהגרפים הם עתיד ה-AI האג'נטי או שהקורס שווה עשרה קורסים בתשלום.

2. מהו knowledge graph: מ-join tables לשאילתת Cypher



השקף Drop the irrelevant records: רשומות product, ו-join table שמחברות את abk ואת ee למוצרים Chair, Lamp, Desk ו-Couch.

Kollegger מתחיל מהמוכר: טבלת persons, טבלת products וטבלת join. השאלה מה ABK קנה דורשת שני joins. השאלה מי עוד קנה את אותם מוצרים דורשת ארבעה. כדי להמליץ על מוצר שאחרים קנו ו-ABK עוד לא קנה צריך joins נוספים, ובלשונו: "There's too many joins. It's getting a little confusing." [4:40].

ההמרה לגרף נעשית בשלושה צעדים: משמיטים את הרשומות שלא רלוונטיות לשאילתה, הופכים את רשומות ה-join לחצים ישירים מאדם למוצר, ומסדרים מחדש. כך רואים מיד ש-ABK ו-EE מחוברים לאותם מוצרים, ושל-EE יש מוצר נוסף שאין ל-ABK. ההגדרה שעולה מזה: מסד נתונים שבו גם nodes וגם relationships הם רשומות מלאות. ל-nodes יכולים להיות כמה labels, לכל relationship יש כיוון ו-type אחד, ולשניהם יש key-value properties. [7:41]



שאילתת ההמלצה ב-Cypher מ-`abk:Person` דרך `PURCHASED` למוצרים, משם לאנשים אחרים ולמוצרים שלהם, עם `WHERE` `NOT abk-[:PURCHASED]->otherProducts`, לצד הגרף שהשאילתה מתארת.

Cypher מתוארת כ-SQL שיש לו nodes. pattern matching כתובים בסוגריים עגולים, ו-relationship מסומן כחץ עם type, למשל -[PURCHASED:]. החלק המעניין הוא שהשאלות מתארות גם את מה שאסור שיתקיים: WHERE NOT קובע ש-ABK לא קנה את המוצר המומלץ (הקו המקווקו לספה בשקף). למה זה מתאים ל-GenAI: "And it turns out that this is super nice for mapping to natural language and really convenient for working with Gen AI and LLMs because LLMs are really good at natural language." [7:41] אפשר לשמור באותו מסד chunks של טקסט עם vector embeddings, ולשלב vector similarity search עם pattern matching באותה שאלה [9:15].

3. מקרה הבוחן: root cause לתקלות מוצר ושלושה תת-גרפים

neo4j

DeepLearning.AI

Root-cause analysis: data engineering questions

1. What is the goal of the analysis?

a. understanding possible manufacturing problems in the supply chain

2. What data is available?

a. bill-of-materials (BOM) files from product to supplier

b. user reviews of products, scraped from a website

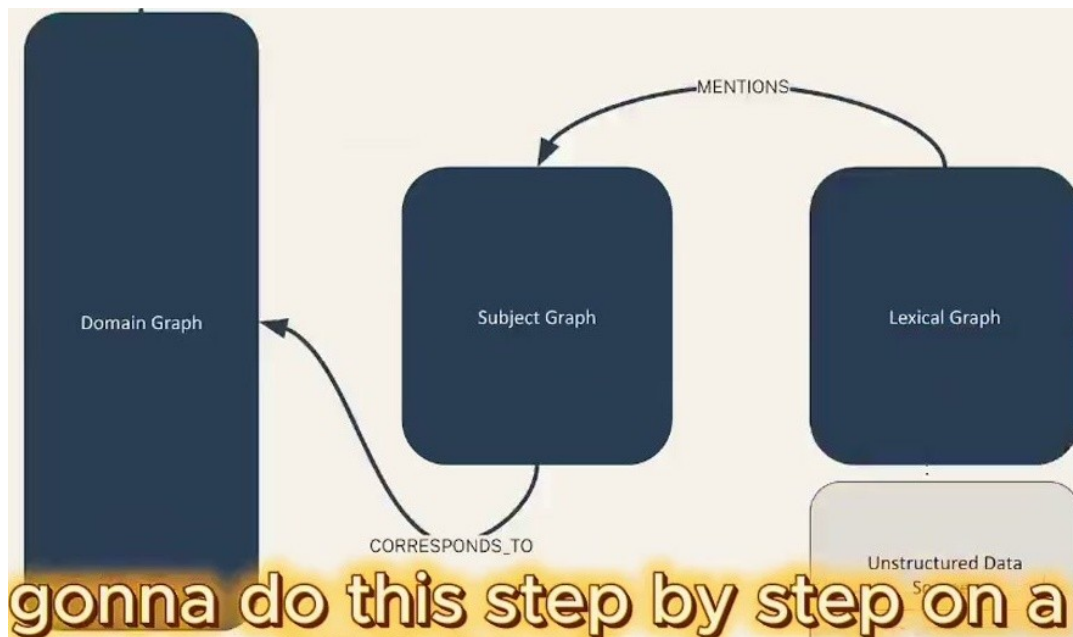
3. How can the data be analyzed?

a. build a knowledge graph connecting all the data

b. query using an agent?

Root-cause analysis: data engineering questions. שלוש שאלות: מה מטרת הניתוח (בעיות ייצור בשרשרת האספקה), אילו נתונים זמינים (קובצי BOM וביקורות משתמשים שנאספו מאתר), ואיך מנתחים (knowledge graph ושאלות דרך agent).

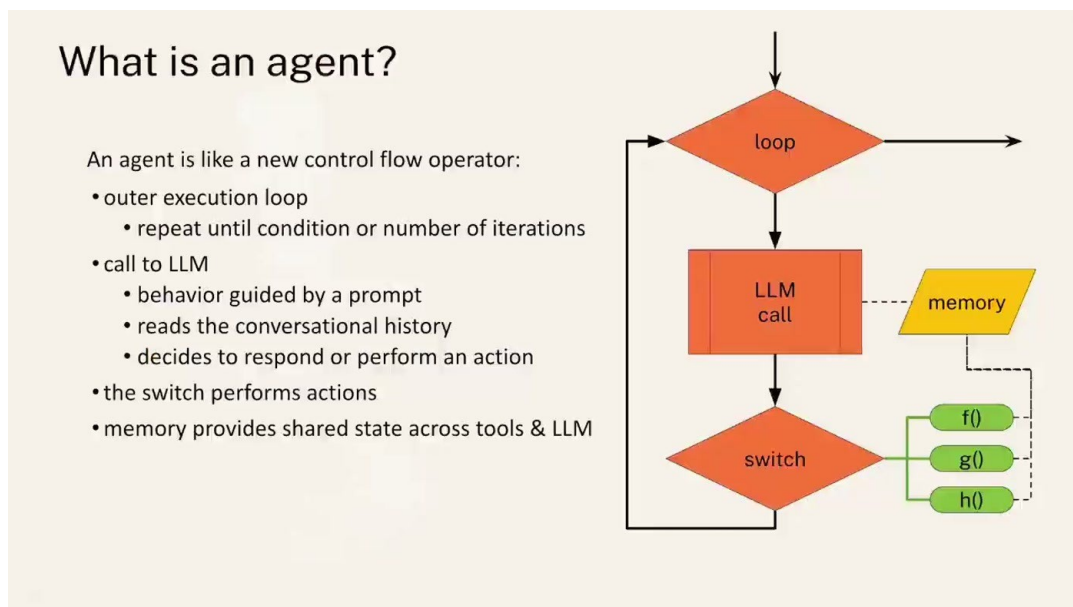
התרחיש: יצרן רהיטים שמקבל תלונות ברשת ורוצה לדעת לאילו מוצרים יש הכי הרבה בעיות, איזה חלק אחראי לבעיה, והאם מקורה בתכנון, בחלק עצמו או בתהליך הייצור. המטרה היא למצוא צעד מתקן: להעביר את הבעיה לצוות התכנון, להחליף ספק או לשפר את התהליך [9:15, 10:49]. Kollegger מציג את זה כעבודה רגילה של data engineer: קודם שואלים שאלות הבהרה על המטרה ועל הנתונים הזמינים. הנתונים הם bill of materials בקובצי CSV, שמוביל מהמוצר דרך parts ו-assemblies עד לספקים, וביקורות משתמשים בקובצי [10:49] markdown.



lexical graph, subject graph, domain graph, unstructured ו-structured מקורות: Domain-Subject-Lexical Graph Model graph, והקשרים MENTIONS ו-CORRESPONDS_TO שמחברים ביניהם.

מודל היעד מחולק לשלושה חלקים שמחברים ביניהם אבל נבדלים זה מזה. ה-domain graph נבנה מה-CSVs ואת המבנה שלו: chunks ו-documents. הוא מיובא ונשאל בדרך אחרת. ביניהם נמצא ה-subject graph: entities שחולצו מה-chunks בצורת subject, predicate ו-object. למשל, ABK loves this table. אם ה-table שהוזכר בביקורת הוא מוצר שקיים בנתונים המובנים, מחברים את ה-entity עד ל-[12:23, 13:55] domain graph. המנגנון המדויק של ההתאמה הזו לא מוסבר בהקלטה.

4. agent ו-multi-agent: ההגדרה ההנדסית



What is an agent?: לולאה, LLM call ו-switch, עם memory שמשותף ל-tools ול-LLM, ופעולות f(), g(), h() ו-f() שה-switch מפעיל.

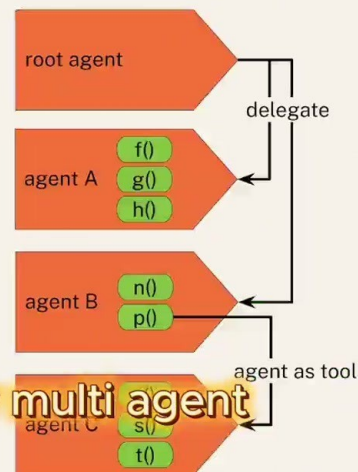
Andreas Kollegger [13:55] I like to think about agents from an engineering perspective. And for — me, it's a fancy new kind of control flow operator

What about multi-agent systems?

Multiple agents collaborating on a goal.

A multi-agent system:

- a hierarchy of agents and sub-agents
- agents specialize in tasks
- agents can "delegate" to each other
- some agents act like tools
- specific checkpoints with the user



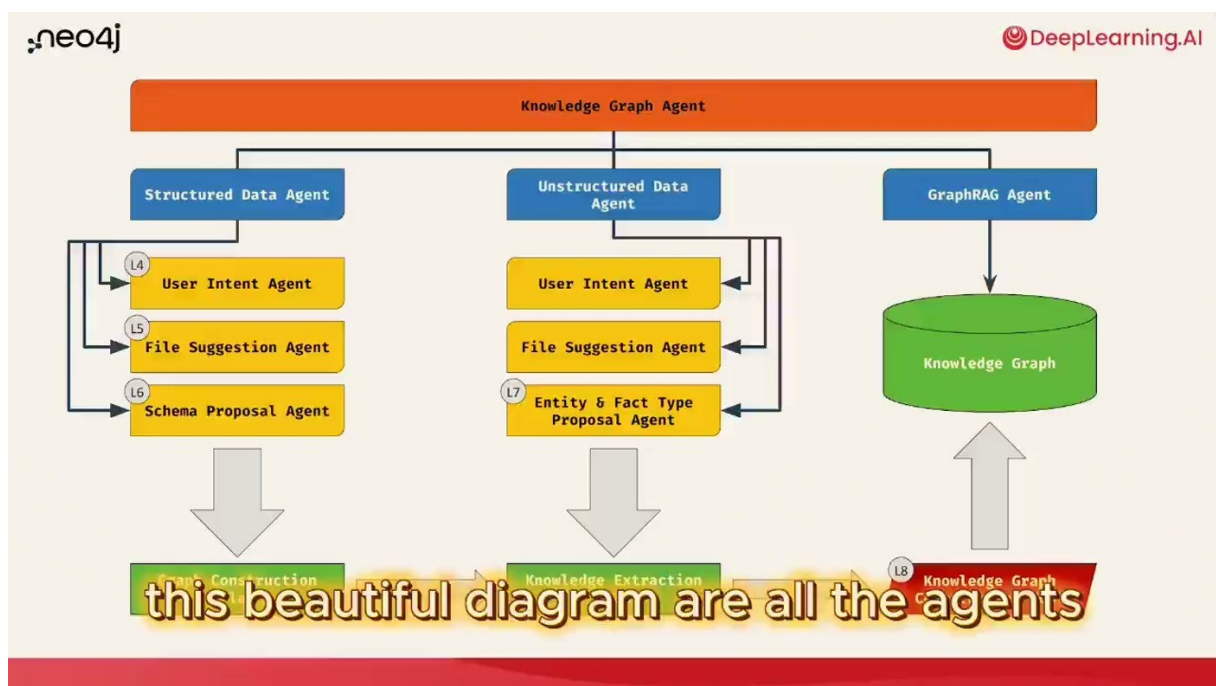
here so what is exactly multi agent system

What about multi-agent systems?: root agent delegate ל-agent A ול-agent B, ובתוך agent B עוד agent שנקרא כ-tool (agent as tool).

לפי ההגדרה הזו, ה-LLM מחליט מה לעשות, אבל הביצוע עדיין קורה בקוד, ב-switch בצד ה-client. היתרונות: reasoning, גישה ל-tools שיכולים לעשות כל מה שקוד יכול, התאמה למצב בעזרת memory, וקלות התחלה כי ההוראות כתובות בשפה טבעית. החסרונות: איטיות, אי-דטרמיניזם, ועלות tokens שמצטברת: "So if you're running an agent over time in production for thousands and thousands of calls, the token cost gets really expensive really quickly." [15:26] לטענתו, multi-agent הוא אחת הדרכים לשפר את החסרונות האלה, אבל הוא לא מסביר איך בדיוק.

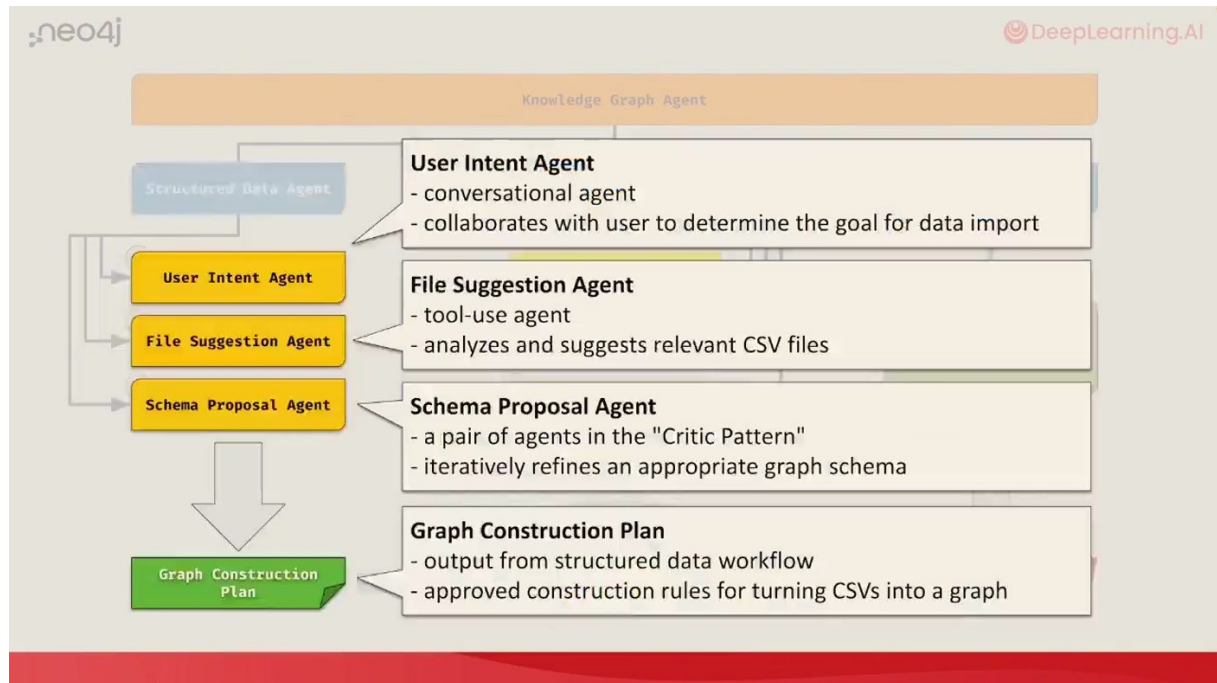
מבחינה מבנית, multi-agent הוא היררכיה: root agent שמנהל את הזרימה, ומתחתיו sub-agents לפי שלבים או משימות. יש שני מנגנונים: delegation, שבו agent מעביר את העבודה ל-agent אחר, ו-agent as tool, שבו agent נקרא כמו פונקציה. בכל מעבר כזה אפשר להוסיף checkpoint מול המשתמש [15:26].

5. ארכיטקטורת המערכת: agents שכותבים תוכניות



מפת המערכת: Knowledge Graph Agent בראש. מתחתיו Structured Data Agent ו-GraphRAG Agent. מתחתם User Intent, File Suggestion, Schema Proposal ו-Entity & Fact Type Proposal, שמזינים את Graph Construction Plan ואת Knowledge Extraction Plan.

ה-Knowledge Graph Agent שבראש הוא agent שיחתי בלבד. הוא לא מבצע עבודה בעצמו, אלא מסביר למשתמש מה אפשרי ומוביל אותו בין השלבים. מתחתיו יש שלוש זרימות. Structured Data Agent, Unstructured Data Agent הם workflow agents שמעבירים עבודה ל-sub-agents, אחד לכל שלב. ה-GraphRAG Agent עונה על שאלות מול הגרף אחרי שהוא נבנה [16:59, 18:31]. Kollegger מציג כל sub-agent כצעד שמהנדס נתונים היה עושה בעצמו. ה-User Intent Agent מבהיר את המטרה, ואת הפלט שלו הוא מגדיר כ-requirements של כל העבודה.



פירוט הזרימה המובנית: User Intent Agent (שיחתי, קובע את מטרת הייבוא), File Suggestion Agent (tool-use, מציע קובצי CSV רלוונטיים), Schema Proposal Agent (זוג agents ב-Critic Pattern) וה-Graph Construction Plan כפלט.

בזרימה המובנית, ה-File Suggestion Agent מסן את הקבצים שבדיסק לפי המטרה, והמשתמש מאשר את הרשימה. אחריו מגיע ה-Schema Proposal Agent, שהוא בעצם זוג agents: אחד מציע schema והשני מבקר אותו, בלולאה. הפלט הוא "It's not a graph itself, but it's a description about how to construct that graph." [20:01]. The goal of this agent is actually to find out what kinds of facts and entities can be extracted, and not do the extraction, but simply describe what's possible." [21:32]. התוצאה היא Knowledge Extraction Plan. בסוף, Knowledge Graph Construction Tool יחיד, שמכיל בתוכו כמה tools, מריץ את שתי התוכניות: בונה את ה-domain graph, מפרק את קובצי ה-markdown ל-chunks, מחשב embeddings, מחלץ facts ו-entities ומחבר אותם לנתונים המובנים. לפי שקף מפת הדרכים, השיעורים בהמשך הם L4 User Intent, L5 File Suggestion, L6 Schema Proposal, L7 Entity & Fact Type Proposal ו-L8 Knowledge Graph Construction [23:06].

6. Google ADK בפועל: agent, tools, delegation ו-state

- **Setup: Agent** מ-Google ADK, מודל OpenAI GPT-4o דרך LiteLLM (בתמלול: GPT-4-0), InMemorySessionService ו-Runner. בדיקת חיבור עם השאלה [23:06, 24:36] are you ready.
- **Neo4j ל-Wrapper**: המודול neo4j_for_adk מכיל helpers בשם tool_success ו-tool_error שמחזירים dictionary עם status, ממיר תוצאות לפורמט שאפשר לשמור (to_python), ו-graphdb כ-singleton שמשותף לכל ה-notebooks [26:07, 27:38].
- **Tool** הוא פונקציה עם say_hello: docstring מקבלת person_name ושולחת שאילתת Cypher עם query parameter (\$). ה-docstring נשלח ל-LLM כתיאור של ה-tool [29:10]. עוד הדגמה: query parameters חוסמים injection, כי ה-payload הזדוני נשאר ערך רגיל של משתנה ולא הופך לחלק מהשאלתה [30:40, 32:11].
- **Agent: name** עם גרסה (hello_agent_v1) לצורך debugging, model, description שמשמש agents אחרים להחליט על delegation, ו-instruction שמשמש את ה-agent עצמו [32:11, 49:00].

- **לולאת ה-events:** ההודעה נאזרת ב-Content עם role=user ו-parts, ואז run_async מחזיר events. היציאה מהלולאה מתרחשת על final response, או על escalate כש-sub-agent לא יכול להמשיך. כל זה נאזר במחלקה AgentCaller, ונוצר דרך factory בשם [38:19, 41:24, 42:56] make_agent_caller.

— **Andreas Kollegger [49:00]** You're gonna spend most of your time, once you've set up any kind of multi-agent system, most of your time is gonna be spent in actually optimizing these instructions

בחלק השני של השיעור נבנה צוות: greeting agent, farewell agent, ו-root agent בלי tools שמקבל אותם ב-sub_agents. ב-instructions של ה-farewell agent יש דוגמאות few-shot לדרכים שונות להיפרד (bye, see). ההבדל מקריאה ל-`tool`: "So the entire conversational history gets passed along and actually-`control of the conversation gets passed along to the sub-agent`." [50:35] במצב verbose רואים את זה בפועל. ה-root מעביר ל-greeting agent דרך `transfer_to_agent`, וזה קורא ל-`say_hello` עם `person_name=ABK`. אחרי זה, כשהמשתמש נפרד, השליטה עדיין אצל ה-greeting agent, וזה מעביר בעצמו ל-farewell agent. אפילו המרצה התבלבל ותיקן את עצמו: "oh actually no, it's still in the control of the greeting sub-agent" [55:09].

session state ב-ADK הוא dictionary ש-ADK עוקב אחרי השינויים בו ושומר עליו עקבי בין כל ה-agents Kollegger [56:40]. מעדיף לגשת אליו דרך `tool_context`: כשהארגומנט האחרון של `tool` הוא `ADK`, `tool_context` מזריק אותו אוטומטית [1:01:17]. האפשרות השנייה היא `output_key`, ששומר את התשובה הסופית של agent ב-`state`. בדמו, `say_hello_stateful` כותב `user_name`, ו-`say_goodbye_stateful` קורא אותו. ה-state מתחיל ריק ומסתיים עם ABK. בהרצה האינטראקטיבית התגלתה בעיה: "Okay, that's kind of funny. So the LLM decided that my name is there based on this message. That's not really great." [1:04:22]. המודל המציא שם לפני שהמשתמש אמר אותו, והבעיה נפתרה רק כשהמשתמש הצהיר במפורש על שמו.

7. מסקנות ופעולות

- **להפריד בין התכנון לביצוע:** תנו ל-agents להפיק תוכנית שהמשתמש מאשר, כמו construction plan או extraction plan, ותנו ל-`tool` דטרמיניסטי לבצע אותה. כך אי-הדטרמיניזם ועלות ה-tokens מוגבלים לשלב התכנון.
- **לעצב קודם את מודל הגרף:** לפני שמחלצים משהו, הגדירו domain, lexical graph, ואת נקודת החיבור בין ה-entities שחולצו לנתונים המובנים.
- **להשקיע ב-description וב-docstring:** הם קובעים את ה-routing. הוסיפו דוגמאות few-shot גם אם נראה שהמודל לא צריך אותן, ותנו לכל agent שם עם גרסה.
- **להריץ verbose לפחות פעם אחת:** כך רואים בפועל מי מחזיק בשליטה אחרי delegation. אל תניחו שהשליטה חוזרת ל-root.
- **לעטוף כל קריאה ל-DB ב-query parameters ובפורמט תוצאה אחיד:** status success או error, כמו `tool_success` ו-`tool_error`.
- **לא לסמוך על ה-LLM שיחלץ entities לבד:** בדמו המודל המציא שם משתמש. צריך validation או checkpoint מול המשתמש לפני ששומרים ב-state.

שאלה פתוחה	למה זה חשוב
איך בדיוק ממפים entity שחולץ מטקסט למוצר ב-domain? graph (CORRESPONDS_TO)?	זה החיבור שמאפשר ניתוח root cause, וההקלטה לא מסבירה אותו.
כמה מדויק ה-GraphRAG בהשוואה ל-RAG רגיל על אותם נתונים?	על זה נשען הטיעון כולו, ולא הוצגו מספרים.
איך multi-agent משפר בפועל את בעיות האיטיות, העלות ואי-הדטרמיניזם?	הטענה נאמרה, אבל המנגנון לא הוסבר.
מה קורה כשה-critic וה-proposer ב-Schema Proposal לא מגיעים להסכמה?	בלי תנאי עצירה מוגדר הלולאה עלולה לרוץ בלי סוף ולצרוך tokens.
איך עוברים מ-InMemorySessionService ל-persistence ב-Neo4j?	זה הוזכר במשפט אחד בלבד [35:17].
איך מונעים מה-LLM להמציא ערכים לפני שהוא כותב ל-state?	בדמו [1:04:22] המודל המציא שם משתמש.

8. מילון מונחים

מונח	פירוש
Knowledge Graph	מסד נתונים שמייצג מידע כ-nodes ו-relationships, ששניהם רשומות מלאות עם properties
Cypher	שפת השאילתות של Neo4j, בנויה על pattern matching (MATCH, WHERE NOT, RETURN)
Join table	טבלת קישור במסד רלציוני שמחברת שתי טבלאות ביחס many-to-many
Domain Graph	תת-הגרף שנבנה מהנתונים המובנים (CSV)
Lexical Graph	תת-הגרף של הטקסט המקורי: documents ו-chunks עם embeddings
Subject Graph	entities ו-facts (subject, predicate, object) שחולצו מה-chunks ומחוברים ל-domain graph
Chunk	קטע טקסט שנחתך ממסמך לצורך embedding ו-retrieval
Root agent / Coordinator	ה-agent העליון שמנהל את הזרימה ומעביר עבודה ל--sub-agents
Delegation	העברת השליטה בשיחה, כולל כל ההיסטוריה, ל-sub-agent
Agent as tool	agent שנקרא כמו פונקציה מתוך agent אחר
Critic pattern	זוג agents, אחד מציע ואחד מבקר, שעובדים בלולאה
Graph Construction Plan	תיאור של איך לבנות את הגרף מה-CSVs. לא הגרף עצמו
Knowledge Extraction Plan	סוגי ה-entities וה-facts שאושרו לחילוך מהטקסט
GraphRAG	מענה על שאלות על בסיס retrieval מתוך הגרף
ADK	Agent Development Kit, ה-framework של Google לבניית agents
LiteLLM	ספרייה שמאפשרת ל-ADK לעבוד מול ספקי מודלים אחרים, כאן OpenAI
Runner / Session	רכיבי הריצה ב-ADK: ה-runner מריץ את ה-agent בתוך session שיש לו state
tool_context	פרמטר ש-ADK מזריק ל-tool ונותן לו גישה ל-session state
output_key	הגדרה ששומרת את התשובה הסופית של agent ב-state
Escalate	סימון של sub-agent שהוא לא יכול להמשיך ושצריך להעביר את הטיפול הלאה
Query parameter	ערך שמועבר לשאילתה כמשתנה (name\$) ולא בשרשור מחרוזות. מגן מפני injection