מאג'נט בודד למערכת אג'נטית בפרודקשן

AWS Summit Tel Aviv 2026 ,סיכום סשן — ARC301

Hannah Karlsbrun, Senior Solutions Architect, AWS | Eran Brown, Principal GenAI Engagement Manager, AWS | Itsik David, Head of R&D, Amdocs Studios division

10 בספטמבר 2026 | משך: כ-42 דקות | הופק מהקלטת הסשן

סיכום מאת קובי אלמוג

על המסמך הזה

המסמך מסכם את הסשן ARC301 — "Build, deploy, and operate agentic applications on AWS Serverless" — מתוך AWS Summit Tel Aviv 2026, מסלול Architecting on AWS. הוא נבנה מתמלול מלא של ההקלטה ומהסליידים שחולצו מהווידאו, כך שאפשר לחזור לתוכן בלי לצפות שוב בארבעים ושתיים הדקות. חותמות הזמן בגוף המסמך מפנות לנקודה המדויקת בהקלטה.

הסשן מחולק לשלושה חלקים ברורים: Hannah Karlsbrun בונה את המערכת האג'נטית מאפס (דקות 0–15), Eran Brown לוקח אותה לפרודקשן (דקות 15–31), ו-Itsik David מ-Amdocs מספר על פרויקט אמיתי שרץ בגישה הזאת (דקות 31–40).

איך לקרוא את זה

- **פרק 1 — תקציר מנהלים:** חמש דקות קריאה. מה נאמר, ומה מזה בר-יישום.
- **פרקים 2–6 — הבנייה:** מאג'נט אחד לארבעה, Strands, MCP, ותקשורת בין אג'נטים ב-A2A.
- **פרקים 7–9 — הארכיטקטורה והאיכות:** שלוש תבניות אורקסטרציה, Event-Driven Architecture, ו-evaluations.
- **פרקים 10–12 — ההרצה:** AgentCore, ה-Managed Harness, והדאטה כיתרון תחרותי.
- **פרק 13 — המקרה של Amdocs:** מודרניזציה של 400 אפליקציות, והמספרים שדווחו מהשטח.
- **פרק 14 — מה לוקחים מכאן:** מסקנות ונקודות להמשך.

> **הערת מקור** המסמך הופק מתמלול אוטומטי של ההקלטה. התמלול בעברית מדויק בקטעים המקצועיים, אך מונחים לועזיים מופיעים בו בשיבוש פונטי ("סטרנץ" = Strands, "אג'נט קור" = AgentCore, "סידאר" = Cedar) — הם נורמלו כאן למונח האנגלי מול הסליידים. הציטוטים אומתו מול ההקלטה בחותמת הזמן המצוינת, ומובאים כפי שנאמרו. שמות ותפקידי הדוברים נלקחו מסליד הפתיחה. מספרים שהופיעו על הסליידים או נאמרו בעל פה צוינו ככאלה ולא הורחבו.

1. תקציר מנהלים

הסשן לא ניסה לשכנע שאג'נטים עובדים. הוא לקח כמובן מאליו שיש לכם אג'נט על הלפטופ, ועסק כולו בשאלה מה עומד בין הדבר הזה לבין מערכת שרצה בפרודקשן — פירוק נכון, ממשקים סטנדרטיים, תבנית אורקסטרציה מתאימה, מדידה, ותשתית הרצה.

- **הטענה המרכזית: זה לא עולם חדש, זו אותה הנדסת תוכנה.** פירוק אג'נט-מונוליט לארבעה אג'נטי-דומיין הוצג כ-decomposition קלאסי, והיתרונות של Event-Driven Architecture הוצגו במפורש כ"שום דבר מפתיע" ביחס למיקרו-שירותים [20:13].
- **הממשקים כבר תוקננו — MCP לכלים, A2A בין אג'נטים.** MCP הוצג כפרוטוקול שרוב התעשייה כבר משתמשת בו לקריאה לכלים [10:50]; A2A הוצג כפרוטוקול שגוגל יצרה במיוחד לממשק בין אג'נטים, כי בין שני אג'נטים אין יחסי שרת-לקוח ברורים [12:22].
- **שלוש תבניות אורקסטרציה, לא אחת.** Orchestrator מכסה לפי Eran Brown את "60-70% מהאמצע" [15:39]; Pipeline לתהליך שידוע מראש; Swarm רק במקרי קצה — "אל תרוצו לסוורם" [18:40].
- **Evaluations הן התנאי לפרודקשן, לא שלב QA.** צוות ה-GenAI Prototyping ב-AWS מדווח על 80% מהפרויקטים שלו שמגיעים לפרודקשן, ומייחס זאת ל-evaluations מיום ראשון; בלוג של AWS שצוטט מדבר על 55% מזמן הפיתוח שמושקע ב-evaluations [21:43].
- **ההמלצה התשתיתית: AgentCore ולא קונטיינר גנרי.** ההבדל שהוצג מול EKS אינו הרנטיים אלא מה שעוטף אותו — memory, gateway לכלים, browser, code interpreter, identity, policy ו-observability כשירותים מנוהלים [24:46, 26:19].
- **המקרה של Amdocs נותן מספרים.** מודרניזציה של 400 אפליקציות בשנה, דרישת לקוח ש-60% מהתהליכים ירוצו אג'נטית, ותוצאות מדווחות של ~35% ירידה בזמן דליברי ו-~70% מהתהליכים שעובדים אג'נטית [30:59, 38:47, 40:18].

מה רלוונטי לארגון גדול

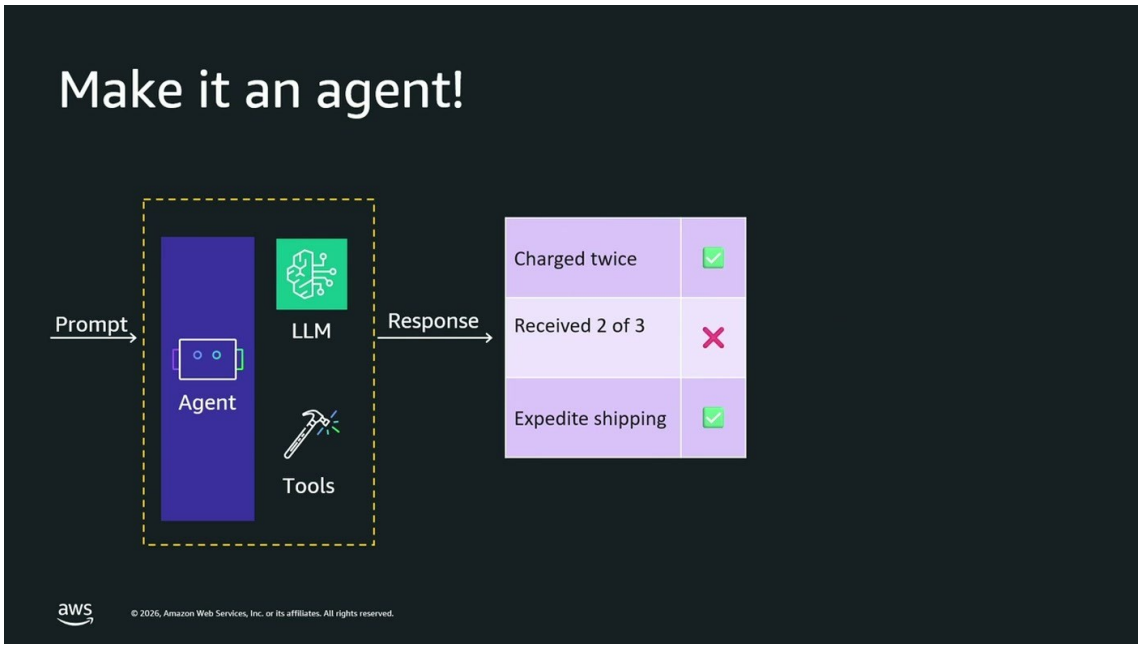
שני חלקים מהסשן ישימים ישירות בארגון מפוקח. הראשון הוא Agent Core Policy: שכבת כללים דטרמיניסטית שיושבת מחוץ לאג'נט ולכן אינה ניתנת לתקיפה דרך prompt injection [26:19] — זו התשובה הארכיטקטונית לשאלה "איך אני מבטיח שהאג'נט לא יחרוג". השני הוא הפצת ה-identity של המשתמש דרך הרנטיים והגייטוויי אל כל ה-tool calls, כך שאג'נט לא יכול להחזיר מידע שלמשתמש עצמו אין אליו הרשאה [26:19]. שניהם הוצגו כחלק מהפלטפורמה ולא כקוד שצריך לכתוב.

2. נקודת הפתיחה: תקלה אחת, שלוש בעיות

Hannah Karlsbrun פתחה בתרחיש אחד שליווה את כל ההרצאה: לקוח קנה שלושה פריטים באתר e-commerce, חויב פעמיים, קיבל שניים מתוך שלושה, ואת השלישי הוא צריך מחר למתנה [0:00]. השאלה לא הייתה איך פותרים את זה, אלא איך עוזרים לנציג השירות לפתור את זה מהר.

המצב ההתחלתי הוא מצב שכל ארגון מכיר: "הסייט שלנו של e-commerce לא נולד אתמול, סביר להניח שיש מערכות קיימות" [1:31] — מודול הזמנות, מודול מלאי, מודול תשלומים, מודול תמיכה. כל אלה לא הולכים לשום מקום. האתגר של הנציג הוא שלושה: לחפש מידע על אותה עסקה בכמה מערכות שונות, לעמוד בדחיפות, ולהיות בקיא בכל נהלי החברה [1:31].

הצעד הראשון המתבקש — אג'נט אחד שמקבל את הבעיה כ-prompt, פונה ל-LLM, מפעיל כלים ומחזיר תשובה [3:09]. אלא שהוא נתקע בדיוק במקום שאפשר לצפות לו.

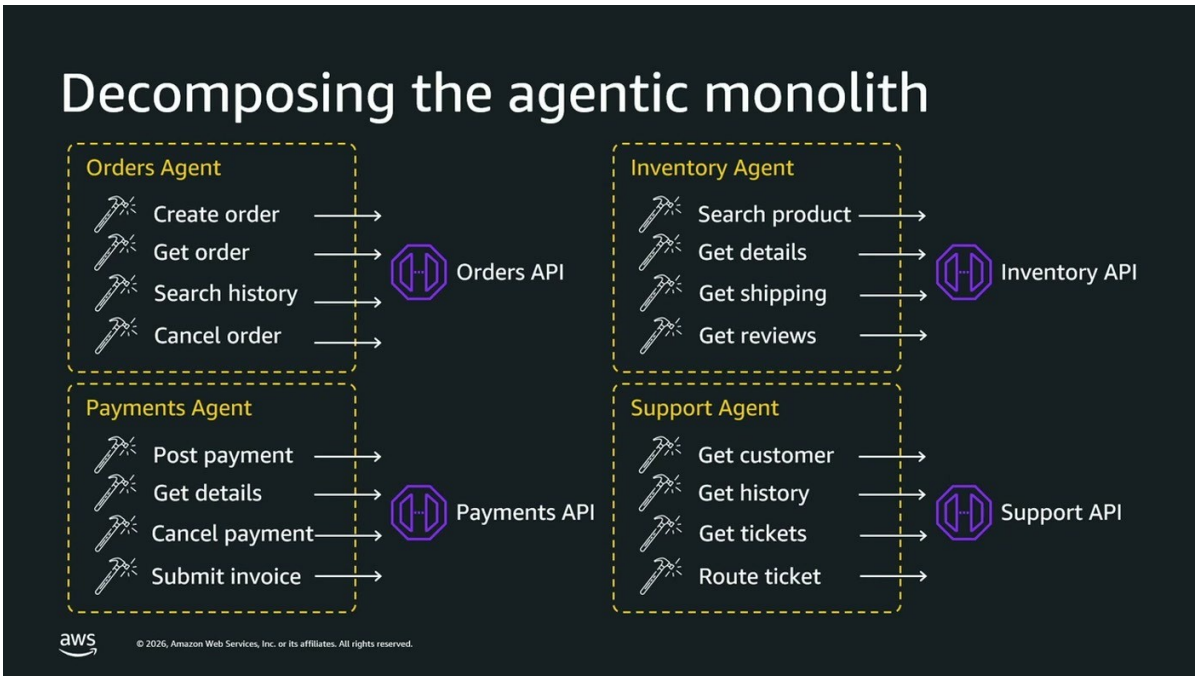


אג'נט בודד מול שלוש הבעיות של אותו לקוח: שתיים סומנו כפתורות, השלישית נכשלה

ההסבר לכישלון הוא ניהולי לא פחות מטכני: "אם יש אג'נט בודד שהוא מטפל בכל מכלל הבעיות שיש לאותו e-commerce, אז זה בטח אג'נט שהוא עם הרבה מאוד system prompt, וזה הופך את האג'נט שלנו לקצת מאבד focus." [3:09] התוצאה, לדבריה: עבודה איטית יותר, prompt גדול יותר ל-LLM, ועלות גבוהה יותר [4:40].

3. פירוק המונוליט האג'נטי

הפתרון שהוצג הוא בדיוק הפתרון שהיה מוצג לפני GenAI: "בדיוק באותה צורה כמו שעשינו במערכות שהן לפני GenAI. שום דבר לא חדש" [4:40]. מחלקים את האג'נט לארבעה אג'נטים קטנים, כל אחד ממוקד בדומיין שלו — הזמנות, תשלומים, מלאי ותמיכה.



ארבעת אג'נטי הדומיין וה-APIs הקיימים שכל אחד מהם עוטף — Orders, Payments, Inventory, Support

שני יתרונות הוצגו, והם מגיעים בסדר. הראשון הוא שאין כאן בנייה מאפס: "האג'נט שלנו יוכל בצורה טבעית מאוד להשתמש ב-API שכבר קיימים לי בארגון... הוא יקרא ל-API הקיימים בתור כלים, בתור [4:40] "tools. השני מגיע רק אחר כך — הוספת יכולות שמבוססות על היכולות של ה-GenAI עצמו, למשל אג'נט מלאי שיודע להמליץ על מוצר חלופי כשהמוצר שהוזמן חסר [4:40, 6:14].

ההפרדה הזאת היא גם מה שמאפשר את כל מה שבא בהמשך: אג'נט שמטפל בדומיין אחד קטן יותר, מהיר יותר, וקל בהרבה למדוד.

4. איך בונים אג'נט: Strands Agents

לפיתוח האג'נטים עצמם הוצג agentic framework. Karlsbrun הזכירה שיש כמה בתעשייה — LangGraph, LangChain ואחרים — והציגה את Strands Agents: פריימוורק שנולד ב-AWS ככלי פנימי לצוותי AWS, שוחרר כקוד פתוח, ותומך ב-Python וב-TypeScript [6:14]. על קצב האימוץ היא אמרה: "תראו שהוא תופס תאוצה, יש למעלה מ-50 מיליון הורדות" [6:14].

אג'נט מחשבון בארבע שורות Python — import, כלי מובנה, יצירת Agent, קריאה

מהדוגמה הנאיבית עברה Karlsbrun לאג'נט ההזמנות האמיתי — "שימו לב, 114 שורות קוד" [7:45]. שלושת המרכיבים שם: כל פונקציית Python הופכת לכלי בעזרת decorator של tool; המודל מסופק כ-endpoint (בדוגמה — Bedrock, ובמפורש "אם אתם לא רוצים להשתמש ב-Bedrock, יש לכם החופש לשים פה endpoint של מודל שרץ בכל model provider. אתם לא מוגבלים" [7:45]); והאג'נט נוצר ממודל ורשימת כלים.

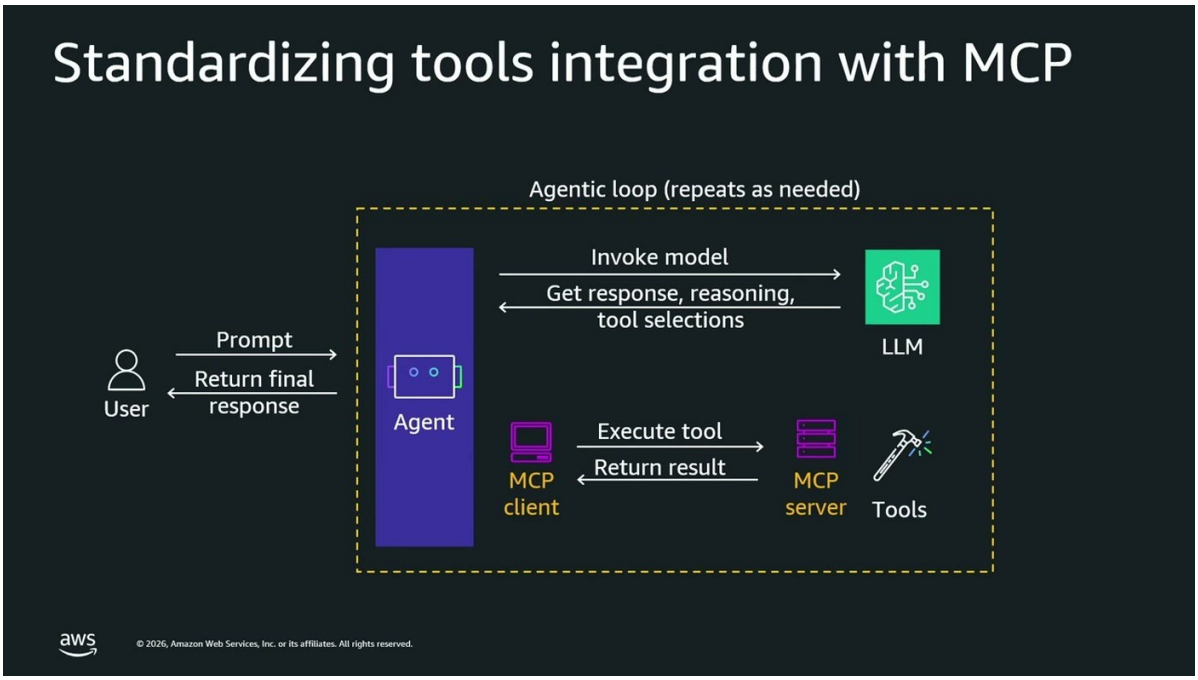
הנקודה החשובה ביותר בפרק הזה היא מה שלא נמצא בקוד: "מה שאין לנו פה זה איזשהו מימוש של קוד פונקציונלי if, then, else — הלקוח קנה, מה עושים? כל זה ה-LLM עושה בשבילנו" [9:16].

מה הפריימוורק נותן

- **טיפול בשגיאות ו-retries מובנה,** יחד עם ה-best practices המקובלות כיום בתעשייה [9:16].
- **אגנוסטיות למודל ולרנטיים:** "כתבתם אג'נט ב-Strands, אתם יכולים להריץ אותו כמובן על AWS בצורה קלה, אבל גם על המחשב האישי שלכם או ב-on-prem compute שרץ אצלכם" [9:16].
- **ה-agentic loop עצמו:** "אותו לב של agent שהוא פונה ל-LLM, התשובה חוזרת, הוא קורא ל-tools, התשובה חוזרת, הוא מסתובב בלופ הזה עד שלא מקבל תשובה מיטבית ומחזיר ל-client שלו" [9:16].

5. כלים דרך MCP

אם האג'נט מפעיל כלים, השאלה היא איך. התשובה שהוצגה כמקובלת בתעשייה היא Model Context Protocol: "אני חושבת כבר כולם מכירים את זה, ורוב התעשייה משתמשת ב-MCP, פרוטוקול סטנדרטי כדי לאפשר לאג'נט לקרוא לכלים" [10:50].

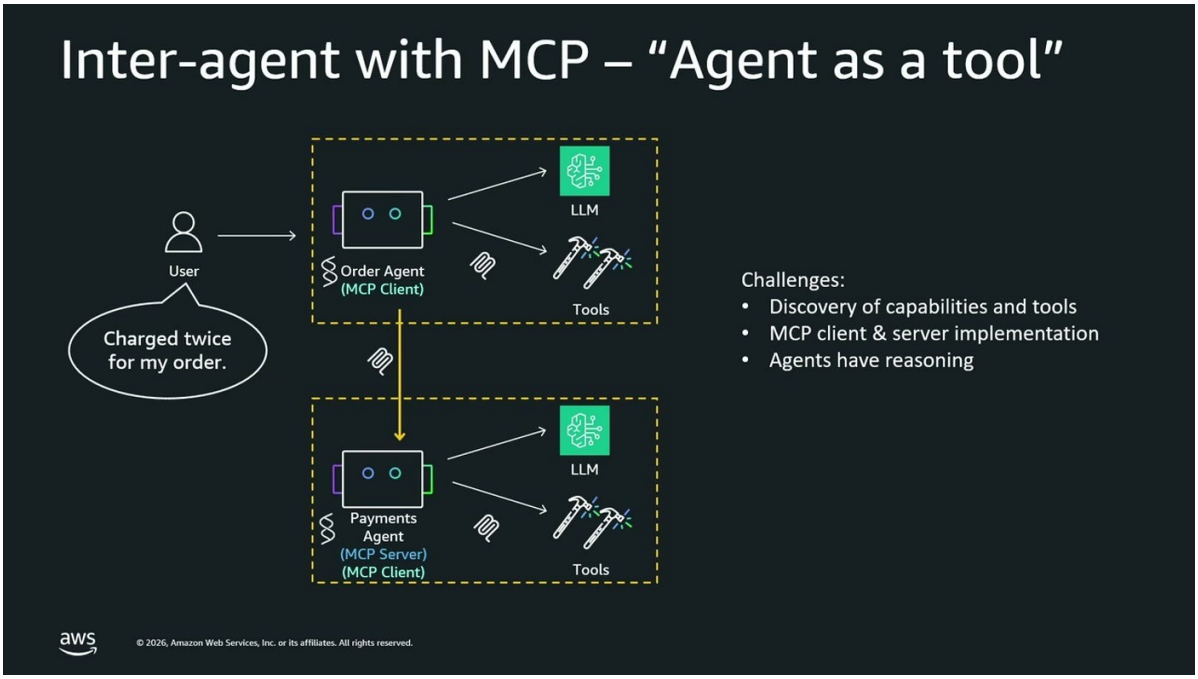


ה-agentic loop המלא: prompt מהמשתמש, קריאות למודל, ו-MCP client/server בין האג'נט לכלים

היתרון המעשי הוא הרכבה: "אנחנו בקלות יכולים להביא כלים שאנחנו כתבנו, כלים שקבוצה אחרת בארגון כתבה ואנחנו רוצים להשתמש בהם, או בכלל MCP צד שלישי כמו Jira, Slack או Teams" [10:50]. במונחי קוד, הפיכת אג'נט Strands ל-MCP client דורשת ייבוא של streamable HTTP client, מתן endpoint של ה-MCP server, ויצירת האג'נט מעליו [10:50, 12:22].

6. איך אג'נטים מדברים ביניהם: מ-Agent as a Tool ל-A2A

ברגע שפירקנו לארבעה אג'נטים, הם חייבים לדבר. האפשרות הראשונה היא פשוט לחשוף אג'נט כ-MCP server, כלומר כ"כלי" של אג'נט אחר: "זה פרקטיקה מקובלת בתעשייה, והיא נקראת Agent as a Tool" [12:22].



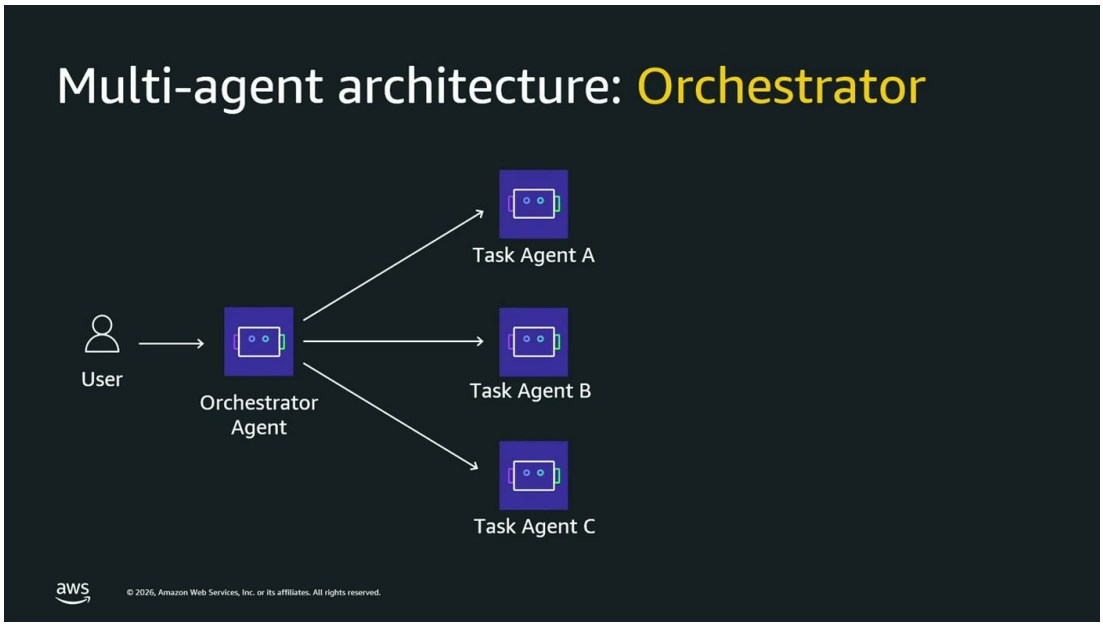
אג'נט התשלומים כ-MCP server שאג'נט ההזמנות קורא אליו, ולצדו שלושת האתגרים שהגישה מייצרת

שני האתגרים שהוצגו הם מהותיים ולא טכניים. הראשון — אסימטריה שלא קיימת: "כשאג'נט קורא לכלי, יש פה עניין של server-client. שני אג'נטים כשהם מדברים ביניהם, אין פה בעצם server ו-client" [12:22]. השני — "אג'נטים יש להם reasoning, הם פונים, הם משתמשים ב-LLM, לעומת tools" [13:54].

מכאן ההסבר למעבר של התעשייה ל-A2A — פרוטוקול שלפי Karlsbrun "המציאו, גוגל בעצם, המציאה במיוחד בשביל ממשק בין האג'נטים" [13:54]. הזרימה שהודגמה בין אג'נט ההזמנות לאג'נט התשלומים: discover capabilities, החזרת Agent Card עם היכולות, בניית task, שיח דו-כיווני אם חסר מידע, והחזרת תשובה [13:54].

7. שלוש תבניות אורקסטרציה

כאן עלה Eran Brown, וניסח את השאלה שמופיעה ברגע שיש ארבעה אג'נטים: "עם אלה ארבעת חברי הצוות, מי מנהל את העבודה? מי מחליט מתי העבודה הסתיימה?" [15:39]. שלוש תשובות הוצגו, לפי רמת מורכבות.



התבנית הראשונה — אג'נט חמישי שמקבל את תפקיד מנהל העבודה ומפנה לאג'נטי המשימה

Orchestrator — ברירת המחדל

בתרחיש הלקוח, האורקסטרטור פונה ל-Payments Agent לבדוק את החיוב הכפול, ל-Orders Agent להבין למה ההזמנה לא סופקה, ולבסוף ל-Inventory Agent כדי לנסות לשלוח פריט או להציע חלופה [15:39]. על היקף ההתאמה Brown היה מפורש: "אם אתם חושבים על ה-100%, לא יודע, 60-70% מהאמצע, הפאטרן הזה מכסה ממש טוב" [15:39].

היתרונות: כל אג'נט אוטונומי ומטפל בדומיין אחד; החלוקה טבעית לבני אדם ולכן קלה לתחזוקה; ו-fault isolation — "אם agent B ייפול, האורקסטרטור יקרא לו עוד הפעם" [17:10]. ה-trade-offs שהוצגו לצדם: שכבת תקשורת נוספת שעלותה אינה אפס; מצבי קצה שבהם "האורקסטרטור יכול להפוך להיות צוואר בקבוק" [17:10]; וקוד ה-fault isolation עצמו, שמרוכז אמנם במקום אחד אבל עדיין צריך תחזוקה [17:10].

Pipeline — כשהתהליך ידוע מראש

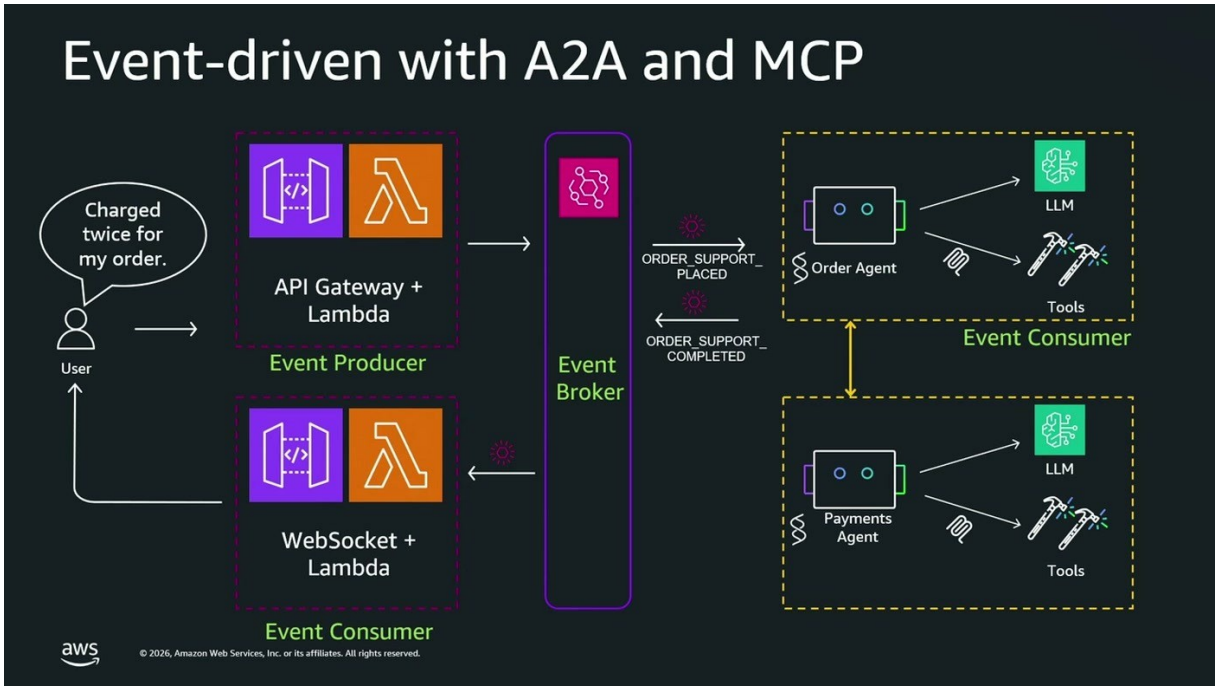
"בפאטרן שנקרא pipeline אנחנו יודעים מראש איך התהליך נראה מבחינת מה צריך לעשות מתי, אבל בכל שלב אני רוצה שלאג'נט שלי תהיה אוטונומיה לבצע פעולות" [17:10]. הדוגמה: יצירת press release או white paper — שלב ראשון איסוף ממקורות הידע, שלב שני פרמול למסמך, שלב שלישי החלת שפת השיווק של החברה. בתוך כל שלב יש reasoning ואוטונומיה, אבל סדר השלבים קבוע [18:40].

Swarm — רק כשצריך

התבנית השלישית היא הפחות דטרמיניסטית: כל אג'נט יכול לטריגר כל אג'נט אחר. המקרה הקלאסי הוא deep research. האזהרה של Brown הייתה חד-משמעית: "איך האג'נט הזה יודע מתי לסיים את העבודה שלו? אין פה אורקסטרטור שאומר סבבה, סיימנו. אין פה pipeline ידוע מראש" — ולכן "אל תרוצו ל-swarm, למרות שזה buzzword שכולם מדברים עליו, תבחרו אותו רק כשהוא באמת מביא את ה-value שהוא צריך" [18:40]. הוא גם הצביע על העלות הנלווית: כמות ה-observability שנדרשת כדי לנתח תקלות בסטאפ כזה.

8. Event-Driven Architecture לאג'נטים

השאלה הבאה הייתה איך מחברים את האג'נטים על גבי תשתית serverless. Brown הצהיר שהשאלה "האם אפשר" אינה מעניינת — "התשובה היא בגדול כן" — ועבר מיד לשאלה למה כדאי [20:13].



המימוש שהוצג: API Gateway ו-Lambda כ-event producer, event broker במרכז, ואג'נטי הזמנות ותשלומים כ-consumers

ארבעת היתרונות הוצגו כולם כמוכרים מעולם המיקרו-שירותים — "שום דבר מפתיע" [20:13]:

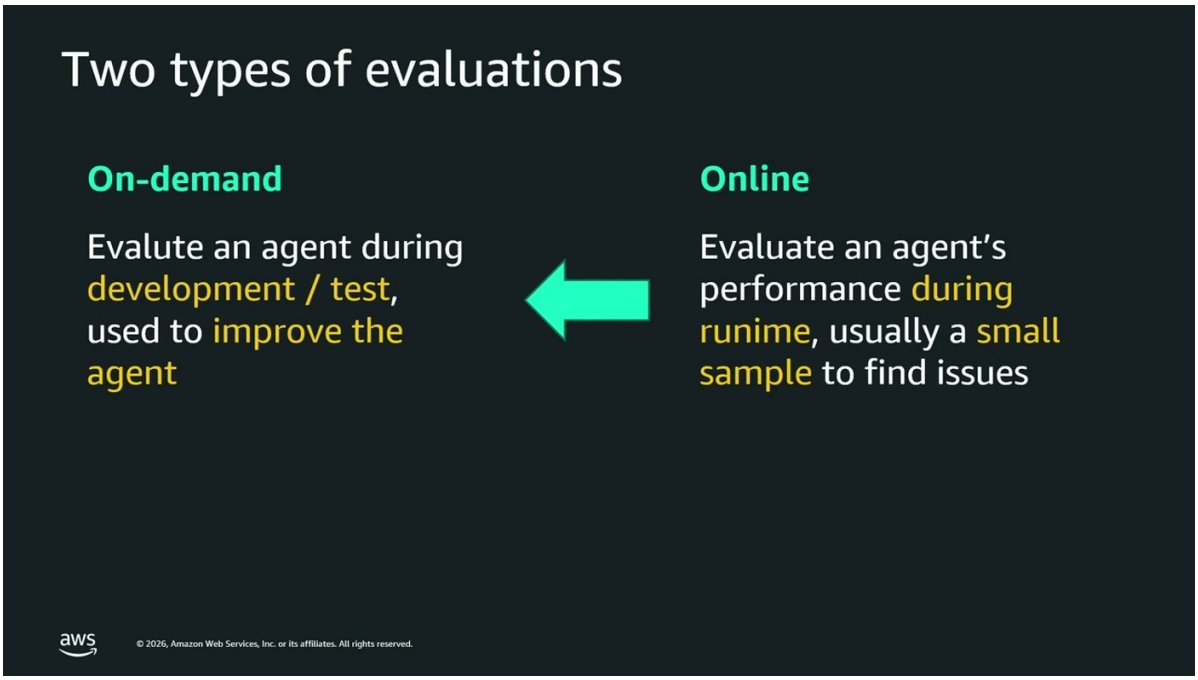
- **Loose coupling.** "כל אג'נט הוא אוטונומי ומתפקד בנפרד, אין לו dependencies באג'נטים אחרים" [20:13].
- **סקיילינג עצמאי לכל אג'נט.** "אם כרגע בדוגמה שלנו יש לי אלף פניות על בעיות inventory ושתי פניות על בעיות של orders, אני לא רוצה להחזיק אלף אג'נטים מכל סוג" [20:13].
- **אינבוקציה א-סינכרונית.** אי-תלות ב-latency, בלי להריץ משאבים idle שמחכים לקריאה הבאה — וכנגזרת, אפשרות ל-replay של event כשמשאב כושל [20:13].
- **Dynamic agent subscription.** כשאג'נט צובר יכולות חדשות לאורך זמן, הוא יכול להירשם לסוגי event נוספים בלי לשנות את המערכת סביבו [21:43].

9. Evaluations — הדבר שיוצא מהסשן הזה

Brown עצר במפורש את הזרימה לפרודקשן: "יאללה מוכנים לפרודקשן? לא ממש, ממש לא" [21:43]. הנימוק שלו הוא גם המשפט שביקש שייקחו מהחלק שלו.

> **Eran Brown, [21:43] —** אם תיקחו רק דבר אחד מהחלק שלי בסשן הזה, קחו את הדבר הזה: אג'נטים הם לא דטרמיניסטים, ואם אני לא דטרמיניסטי, הבדיקות שלי, ה-evaluations, הופכים להיות קריטיים. קחו קוד פייתוני, תריץ אותו עשר פעמים, אמור לרוץ אותו דבר. אג'נט? לא חייבים evaluations.

לתמיכה בטענה הוא הביא שני נתונים. הראשון מהצוות שהוא מוביל: "אני מרכז את צוות ה-GenAI Prototyping ב-AWS... הצוות שלנו רואה 80% מהפרויקטים שלנו מגיעים לפרודקשן, בניגוד לתעשייה שאצלה הסטטיסטיקה הפוכה. ה-secret sauce שלנו: evaluations מ-[21:43] "1 day. השני מפרסום של AWS: "כשאנחנו כותבים אג'נטים אנחנו משקיעים יותר מחצי מהזמן, 55%, ב-evaluations. קחו את המספר הזה איתכם" [21:43].



שני סוגי ההערכה וכיוון ההזנה ביניהם: ממצא מ-online חוזר פנימה אל ה-on-demand

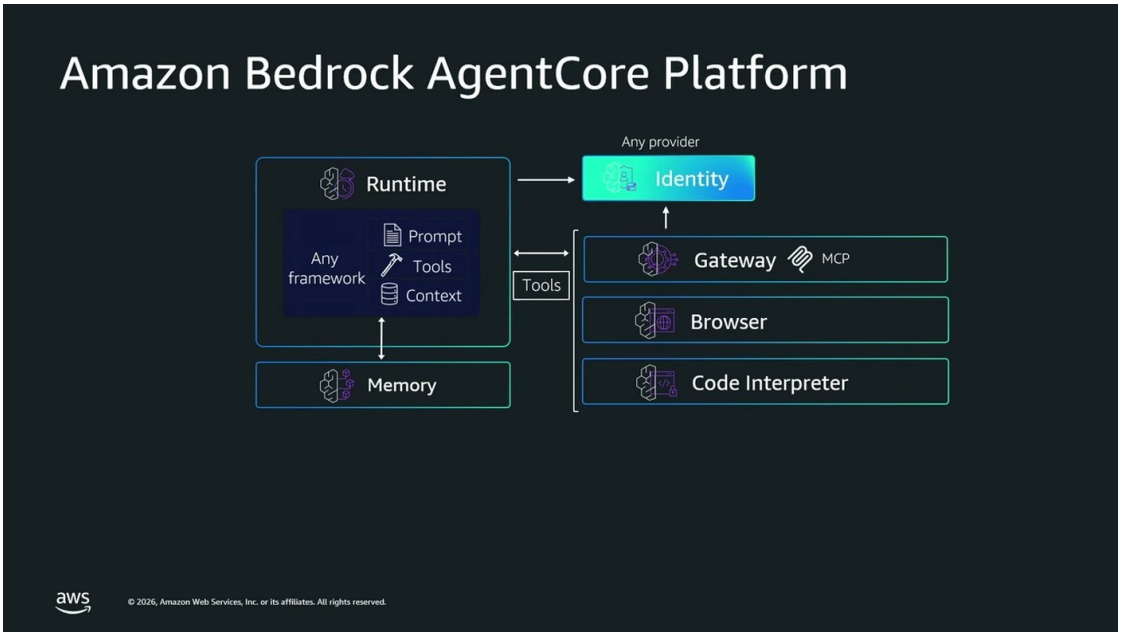
ההבחנה שהוצגה היא בין on-demand evaluations — שרצות במהלך הפיתוח, ב-V1, V2, V3 — לבין online evaluations. ה-on-demand חשובות אבל לא מספיקות: "באוולויואציות האלה אני אחשוף את האג'נט שלי ל-50, 100 סנריות. תגיעו לפרודקשן, הלקוחות שלכם ידאגו להכיר לאג'נט שלכם המון מצבים חדשים" [23:14]. לכן נדרשת דגימה של אחוז או חצי אחוז מהקריאות בפרודקשן, שעליה רץ LLM as a judge.

החלק הקריטי הוא החיבור בין השניים: "כל פעם שבפרודקשן מצאתי משהו שלא עבד טוב, אני חייב להחזיר אותו לתוך ה-evaluation dataset שלי ל-on-demand. אחרת, כמו באג שתיקנתי אותו ולא שמתי עליו בדיקה, הוא יחזור בעוד שלוש גרסאות" [23:14].

10. איפה זה רץ: Amazon Bedrock AgentCore

Brown הקדים הסתייגות חשובה: כל מה שנאמר משלב ההרצה ואילך "נפרד לגמרי ממה ה-agentic framework שבחרתם. זה לא ישנה בכלל" [24:46]. האפשרויות הטבעיות — קונטיינרים מאחורי ALB, או Lambda מאחורי API Gateway — הוצגו כ"אופציות ולידיות לדברים פשוטים"; ברגע שמדברים על סקייל ופרודקשן, ההמלצה היא AgentCore.

> **Eran Brown, [24:46]** בסוף מריץ קונטיינר, אז תגידו מה ההבדל בינו לבין EKS? ש-AgentCore — לא נעצר ברנטיים בקונטיינר, אלא הוא עוטף את הקונטיינר הזה בסט של שירותים שהם מאוד רלוונטיים ספציפית לאג'נטים.



רכיבי הפלטפורמה סביב הרנטיים: Identity, Gateway עם MCP, Browser, Code Interpreter ו-Memory

- **Memory כשירות מנוהל.** "אני קורא לאג'נט שלי כמה פעמים ואני מצפה שהוא יזכור על מה דיברנו" — short term, long term [24:46].
- **Gateway יחיד לכלים.** "אני יודע להגיד מאות tools מאחורי gateway יחיד, וככה האג'נט שלי לא צריך להתבלבל מכמות ה-tools" [24:46].
- **Browser.** לכל מה שאין לו API: "לך לאינטרנט, תקנה לי כרטיס רכבת לשעה שמונה בבוקר מחר — אני צריך browser, כי לא כל אתר חושף API" [24:46].
- **Code Interpreter מבודד.** אם האג'נט מייצר קוד, "אני לא רוצה שהקוד הזה ירוץ בתוך הקונטיינר של האג'נט, הקוד הזה עלול בטעות להיות גם malicious או פשוט unstable" [24:46].
- **Identity שמופצת פנימה.** גם הרנטיים וגם הגייטוויי מתממשקים לשירות ה-identity, כדי ש"כל ה-tool calls שלכם ירשו את ה-identity של ה-user" [26:19].
- **Observability מובנית.** אינטגרציה ל-OpenTelemetry שמגיעה built-in ואינה דורשת בנייה [26:19].
- **Evals כהגדרה.** אפשר להורות ל-AgentCore לשלוח חצי אחוז מקריאות ה-invoke ל-LLM as a judge [26:19].

Agent Core Policy — למה זה מעניין

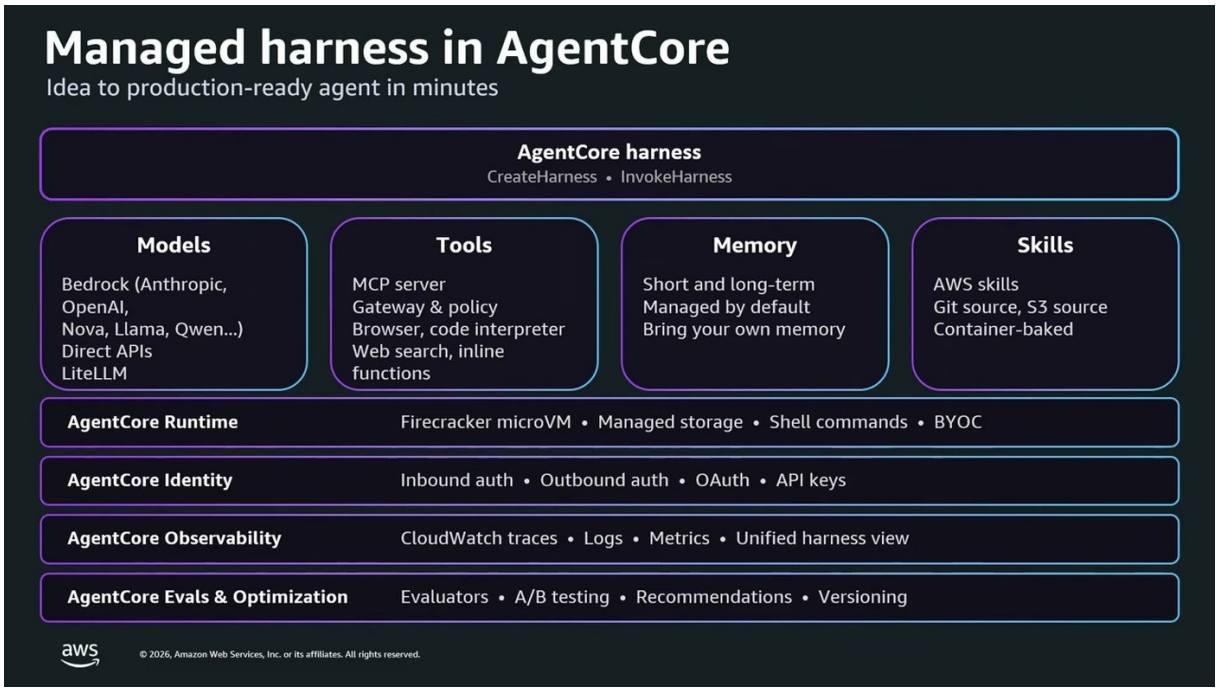
הרכיב הזה קיבל את ההסבר המנומק ביותר, והוא זה שרלוונטי לכל דיון אבטחה. נקודת המוצא: "תזכרו שהאג'נטים עלולים להתבלבל בין ה-system prompt לבין הדאטה שאני מזריק לו תוך כדי. הם לא יודעים להבדיל בין דאטה לבין הוראות" [26:19]. המסקנה הארכיטקטונית: "אני חייב משהו דטרמיניסטי שירחף עליהם, סט של כללים. וזהו Agent Core Policy. הוא משתמש בשפה דטרמיניסטית בשם Cedar כדי להגדיר חוקים לאג'נט, והוא נמצא מחוץ לאג'נט, ולכן לא ניתן באמצעות injection ל-prompt, למשל, לתקוף אותו" [26:19].

הערת תמלול שם השפה נאמר בהקלטה ונקלט כ"SIDAR". בהקשר של מנוע הרשאות ב-AWS מדובר ב-Cedar, שפת המדיניות של Amazon Verified Permissions. הנורמליזציה נעשתה לפי ההקשר; הסליידים לא הציגו את השם.

11. AgentCore Managed Harness

ההכרזה שהוצגה — לפי Brown מסאמיט ניו יורק ביולי [27:55] — נועדה לפתור בעיה ספציפית: מפתח שאינו איש DevOps ואינו מכיר את שירותי AgentCore, ורוצה להשתמש בהם כקופסה שחורה.

"Harness הוא בעצם החיבור בין המודל שבו אתם משתמשים, לכלים, לזיכרון, ל-skills. זה מה שמגדיר את האג'נט שלכם, זה ה-harness שלו" [27:55].



ארבעת עמודי ה-Models, Tools, Memory, Skills — harness — מעל שכבות Runtime, Identity, Observability ו-Evals

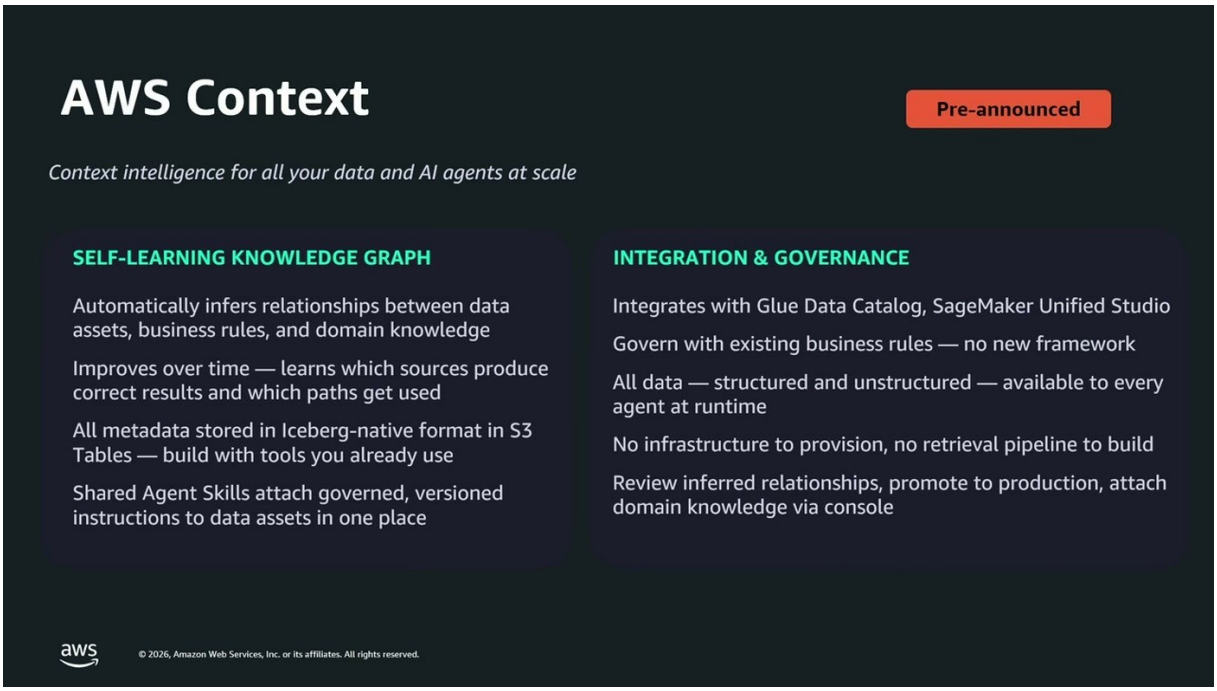
ממשק המפתח הוא שתי פקודות: "בסוף מפתח פוגש שתי פקודות, create harness, invoke harness, בסוף סיפור" [27:55]. שלושת המאפיינים שהוצגו כיתרון הם ניהול כקובץ קונפיגורציה שנכנס ל-git; שימוש בכל הסטאק של AgentCore "בלי שתצטרכו להבין אותו"; ואפשרות export לקוד: "בסוף הפיתוח אני יכול לעשות export לכל מה שבניתי לקוד, להריץ אותו, לעשות commit לגיט שלי, ולנוע מהר מאוד לפרודקשן" [29:27].

הערה שכדאי לקחת מהסשן: Brown ציין ש-"AgentCore הוא אחד השירותים שבשנה האחרונה קיבלו עוד ועוד ועוד יכולות, שווה לעקוב אחרי ההכרזות שלו" [27:55] — כלומר כל תמונת המצב כאן היא נכון לספטמבר 2026.

12. איפה נשאר היתרון התחרותי

החלק האחרון של Brown היה השאלה הכי פחות טכנית בסשן: "אם כולכם יכולים לגשת לכל ארגז הכלים הזה, ואג'נטים מייעלים לכם את התהליכים, איפה אתם מייצרים את היתרון היחסי שלכם?" [29:27]. תשובתו: "לב ליבו של היתרון בסוף הופך להיות דאטה" — הדאטה שכבר יש לכם והדאטה שאתם מסוגלים לאסוף.

החלוקה שהוצגה היא לשלושה סוגים, ולכל אחד נמסר מענה: structured data דרך tools; unstructured data דרך knowledge bases שממפות ויקי, Slack וכדומה; ו-public או live data דרך כלי web search שרץ בלי לצאת מה-VPC שלכם [29:27].

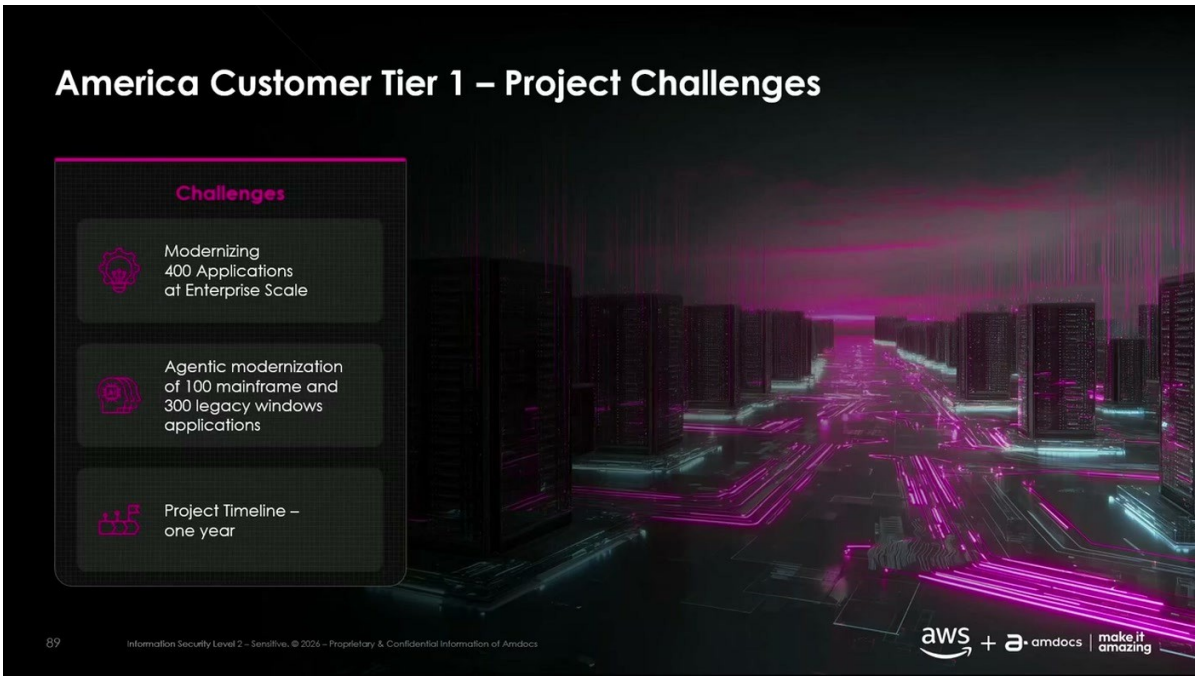


ההכרזה המוקדמת שהוצגה: knowledge graph שנבנה מהדאטה, עם אינטגרציה ל-Glue Data Catalog ול-SageMaker Unified Studio

על השירות הזה Brown ציין במפורש שהוא "עדיין לא שוחרר" [29:27], וגם הסלייד ציין זאת בתווית Pre-announced. מי שבונה עליו תוכנית עבודה צריך לוודא סטטוס עדכני.

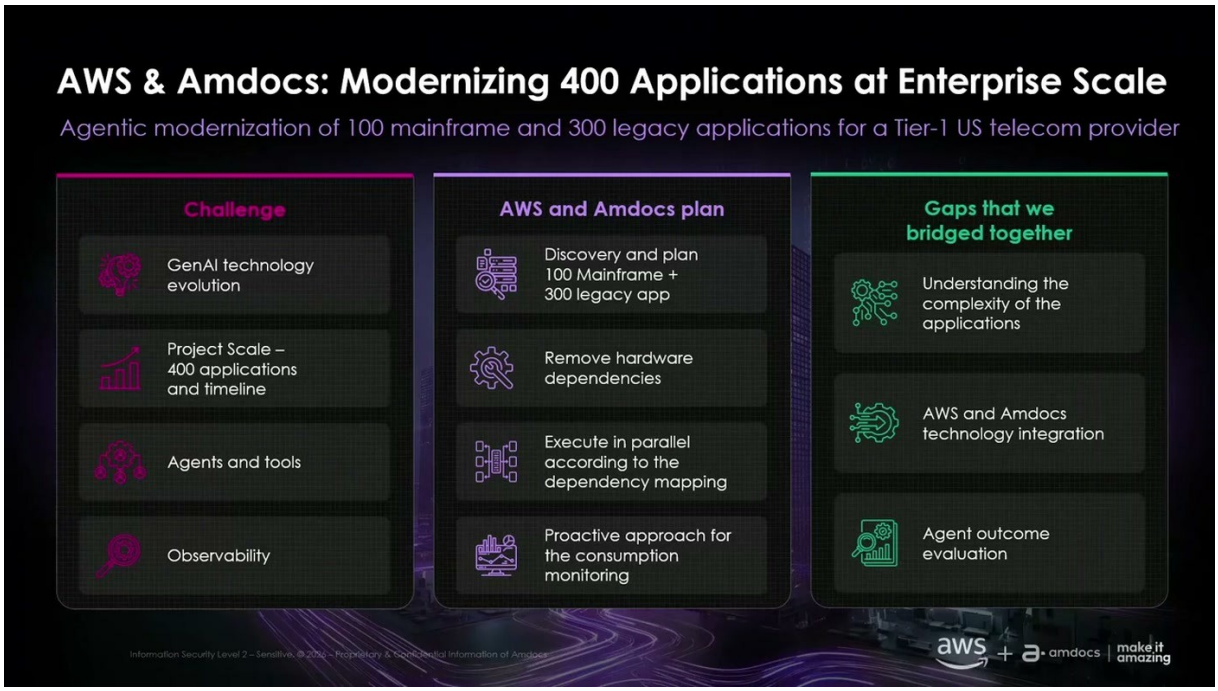
13. המקרה מהשטח: Amdocs ומודרניזציה של 400 אפליקציות

Itsik David, שמנהל את ה-R&D בחטיבת Amdocs Studios, סיפר על אתגר שהתקבל מלקוח גדול בארצות הברית: מודרניזציה של 400 אפליקציות — מהן 100 mainframe ו-300 Legacy Windows Applications — בטיימליין של שנה [30:59].



ארבעת האתגרים כפי שהוצגו: היקף, טיימליין של שנה, אג'נטים וכלים, ו-observability

מעבר להיקף, הלקוח הוסיף דרישות שאינן שגרתיות. אחת מהן היא הדרישה שנשמעת כמו התערבות בבחירה הטכנולוגית: "הוא מבקש שהתהליכים ירוצו בצורה אג'נטית, לפחות 60% מהם" [30:59]. לצד זה: ודאות שכל התהליכים העסקיים ממשיכים לעבוד כמו שהיו, הבנת כל ה-dependencies ברמת הביזנס, ומעקב מתמיד אחרי observability וצריכת טוקנים כדי לא להגיע לכמות מוגזמת [30:59, 32:31].



שלוש העמודות שהוצגו: האתגר, התוכנית המשותפת של AWS ו-Amdocs, והפערים שנסגרו

איך זה נבנה

הפתרון של Amdocs נקרא Agentic Cooperation System — ACS — ומחולק לאזורים; David התמקד ב-IT Services, שמכיל שמונה פילרים: Mainframe Modernization, Experience Design, Data Transformation, Application Modernization, Cloud Migration, Data Platform, Quality Engineering ו-Data Migration. בכל פילר יש workflows שמייצגים את השירות ללקוח, ובכל workflow רצים אג'נטים שמייצרים את ה-outcome [32:31].

התהליך התחיל ב-discovery בעזרת אג'נטים על האפליקציות עצמן, ואחריו planning. ארבע החלטות ארכיטקטוניות שצוינו: שקיפות מלאה מול הלקוח; אפס hardware dependencies, "אנחנו הולכים להריץ כמויות גדולות של אג'נטים ואסור שיהיה משהו שיעצור אותנו"; dependencies mapping של כל האפליקציות והתהליכים העסקיים כדי לאפשר הרצה מקבילית; וגישה פרואקטיבית למעקב אחרי consumption וטוקנים [32:31, 34:02].

המסקנות אחרי שלושה חודשים

אחרי שלושה חודשים בפרויקט נעשתה חשיבה מחדש, ומתוכה יצאו ארבע מסקנות מעשיות [34:02]:

- **סטנדרטיזציה של אג'נטים.** כל אג'נט חייב לעמוד בסט אחיד של security, context, evaluation, guardrails ו-observability.
- **A2A כתנאי לשיתוף פעולה.** "כדי שהאג'נטים ידברו אחד עם השני, היינו חייבים להשתמש בפרוטוקול A2A, ככה שה-workflow שלנו מכיל אג'נטים שהאינפוט של אחד יכול להיות האאוטפוט של השני" [34:02].
- **AgentCore Gateway לניהול כמות הכלים.** "כמות הכלים שיצרנו הולכת וגדלה כדי לתמוך בכל הסוגים של האפליקציות" — ומכאן הצורך בחשיפה מרוכזת ומנוהלת בארגון [34:02].
- **AgentCore Runtime ו-Observability.** האג'נטים רצו ב-AgentCore Runtime, והמידע שנאסף ב-Observability הועבר פנימה למערכות הניטור של Amdocs [35:42].

התהליך בפועל

שרשרת הכלים שתוארה: Amazon Q מספק את ה-insights על ה-dependencies, האפליקציות הקיימות, מספר המשתמשים והעומס — ועל בסיס זה נבחרות האפליקציות להתחיל בהן. אחר כך רצים אג'נטי ה-discovery שיושבים ב-AgentCore, אג'נטי mainframe ואג'נטי cloud. יחד עם AWS Transform מופקים ה-rules וקבצי ה-MD שמזהים אילו תהליכים נדרשים לשדרוג — בדוגמה שהוצגה, שדרוג גרסת Java. משם הכול עובר ל-Kiro, ו-FDE (Forward Deployment Engineer) יושב מול Kiro ומוודא שהארכיטקטורה החדשה מתאימה ושהקוד עובד. אז עולה הכול לסביבת טסטים, שבה Amdocs הוסיפה שכבה גדולה של אג'נטים לאיכות, פרפורמנס וסקיוריטי, והאג'נט האחרון עושה deployment ל-AWS ל-Kubernetes [35:42, 37:15].

בפריסה עצמה: ה-UI, ה-API Gateway והמיקרו-שירותים יושבים ב-Kubernetes; האג'נטים יושבים ב-AgentCore, כולל אג'נטים שעושים wrap ל-AWS Transform; הכלים יושבים על MCP server; ו-"המתווך שלנו בין האג'נטים לבין הכלים, זה ה-AgentCore Gateway" [38:47]. סוגי הכלים שצוינו: open source tools מהשוק, כלים ייחודיים של Amdocs, כלים שהלקוח עצמו יצר, ו-AWS Transform שנקרא דרך MCP.

התוצאות שדווחו

- **ירידה בזמן הדליברי.** "ראינו בעצם ירידה של אזור ה-35% מאיך שהיינו עובדים בעבר, בלי לפגוע בביזנס של הלקוח" [38:47].
- **יותר מודרניזציה באותו זמן.** "יש לפחות 35% יותר מודרניזציה שאנחנו מסוגלים לעשות", תוך שמירה על האיכות של האפליקציות שעוברות את התהליך [38:47].
- **עמידה בדרישת ה-60%.** "רוב התהליכים שלנו, באזור 70%, עובדים לגמרי בעזרת האג'נטים ובעזרת ה-workflow שיצרנו" [40:18].

סייג המספרים בפרק זה הם מספרים שדווחו על הבמה על ידי Amdocs, בפרויקט שעל פי הדובר נמצא "ממש לקראת סוף הפרויקט עצמו" [38:47]. לא הוצגה מתודולוגיית מדידה, בסיס ההשוואה לא פורט, ושם הלקוח לא נמסר (תואר כ-Tier-1 US telecom provider על הסלייד).

14. מה לוקחים מכאן

- **אל תבנו אג'נט אחד גדול.** אותם שיקולים שהובילו למיקרו-שירותים מובילים כאן לאג'נטי דומיין: system prompt קצר יותר, מיקוד, מהירות ועלות. הצעד הראשון הוא לעטוף APIs שכבר קיימים, לא לבנות יכולות חדשות.
- **בחרו תבנית אורקסטרציה במודע.** Orchestrator הוא ברירת המחדל הסבירה לרוב המקרים; Pipeline כשהסדר ידוע מראש; Swarm רק כשהערך מצדיק את חוסר הדטרמיניזם ואת עלות ה-observability.
- **Evaluations הן עבודת תשתית, לא בדיקה.** שתי משפחות — on-demand בפיתוח ו-online בפרודקשן — עם לולאת הזנה חזרה מה-online אל ה-dataset. המספר שהוזכר, 55% מזמן הפיתוח, הוא סדר גודל שכדאי לתכנן לפיו.
- **הפרידו את חוקי המדיניות מהאג'נט.** אג'נט אינו מבחין בין דאטה להוראות. שכבת policy דטרמיניסטית מחוץ לאג'נט היא ההגנה הארכיטקטונית מול prompt injection, ולא ניסוח זהיר יותר של ה-system prompt.
- **הבחירה בין קונטיינר ל-AgentCore היא בחירה במה שעוטף.** memory, gateway, browser, code interpreter, identity, policy, observability ו-evals — אלה הרכיבים שתצטרכו לבנות בעצמכם אם תריצו על תשתית גנרית.
- **הפריימוורק הוא ההחלטה הכי הפיכה.** Brown הדגיש שכל שלב הפריסה נפרד לגמרי מהפריימוורק שנבחר [24:46]. ההחלטות שקשה לחזור מהן הן החלוקה לדומיינים, הפרוטוקולים, ותשתית המדידה.

המשך עצמאי

בסיום הופנו הצופים לשני מסלולי המשך: מסלול מסודר ב-AWS Skill Builder למי שרוצה ramping-up מובנה, וסדנה בגישת learning by building שפיתח הצוות של Brown — "מתחילים בלבנות אג'נט, מסיימים בלהשיק אותו בפרודקשן", עם מרכיב developer ומרכיב DevOps [40:18]. הקישורים הופיעו כקודי QR על הסלייד האחרון.

מילון מונחים

מונח	מה זה, כפי שהוצג בסשן
Strands Agents	פריימוורק קוד פתוח לפיתוח אג'נטים שנולד ב-AWS ככלי פנימי; Python ו-TypeScript; מגיע עם agentic loop, retries ו-error handling מובנים.
Agentic loop	לולאת הליבה של אג'נט: פנייה ל-LLM, קבלת תשובה, קריאה לכלי, קבלת תוצאה — עד שמתקבלת תשובה מספקת.
MCP	Model Context Protocol. פרוטוקול סטנדרטי לקריאה של אג'נט לכלים, שמאפשר לצרוך כלים של צוותים אחרים או של צד שלישי.
A2A	פרוטוקול Agent-to-Agent שנוצר בגוגל לממשק בין אג'נטים: Agent Card, discover capabilities, בניית task ושיח דו-כיווני.
Agent as a Tool	חשיפת אג'נט כ-MCP server כדי שאג'נט אחר יקרא אליו ככלי. עובד, אבל מתעלם מכך שאין כאן יחסי server-client אמיתיים.
Orchestrator / Pipeline / Swarm	שלוש תבניות אורקסטרציה: מנהל עבודה מרכזי, שרשרת שלבים ידועה מראש, וקבוצה שבה כל אג'נט יכול לטריגר כל אחר.
On-demand / Online evaluations	הערכת אג'נט בזמן פיתוח מול תרחישים ידועים, לעומת הערכת דגימה מהקריאות בפרודקשן בעזרת LLM as a judge.
AgentCore	פלטפורמת Amazon Bedrock להרצת אג'נטים: רנטיים בקונטיינר עטוף ב-Memory, Gateway, Browser, Code Interpreter, Identity, Policy, Observability ו-Evals.
AgentCore Harness	הגדרה מנוהלת כקובץ קונפיגורציה של מודל, כלים, זיכרון ו-skills; שתי פקודות — create harness ו-invoke harness — עם אפשרות export לקוד.
Agent Core Policy	שכבת כללים דטרמיניסטית בשפת Cedar שיושבת מחוץ לאג'נט, ולכן אינה חשופה ל-prompt injection.
ACS (Amdocs)	Agentic Cooperation System — הפלטפורמה של Amdocs, מחולקת לאזורים ולפילרים, ובכל פילר workflows שמורכבים מאג'נטים.

מונח	מה זה, כפי שהוצג בסשן
FDE	Forward Deployment Engineer — המהנדס שיושב מול Kiro ומאשר שהארכיטקטורה והקוד שנוצרו עומדים בציפייה לפני מעבר לטסטים.