קומפיוט חסר־מצב, מערכות בעלות מצב: ארכיטקטורת Workflows עמידים

ARC304 — סיכום סשן, AWS Summit Tel Aviv 2026

Uri Segev, Principal Serverless Solutions Architect, AWS · Shani Kazaz, Senior AppMod Specialist, AWS · Efi Merdler-Kravitz, Staff Engineer, Melio

10 בספטמבר 2026 | משך: כ-37 דקות | הופק מהקלטת הסשן

מסלול: Architecting on AWS | רמה: 300 — Advanced

על המסמך הזה

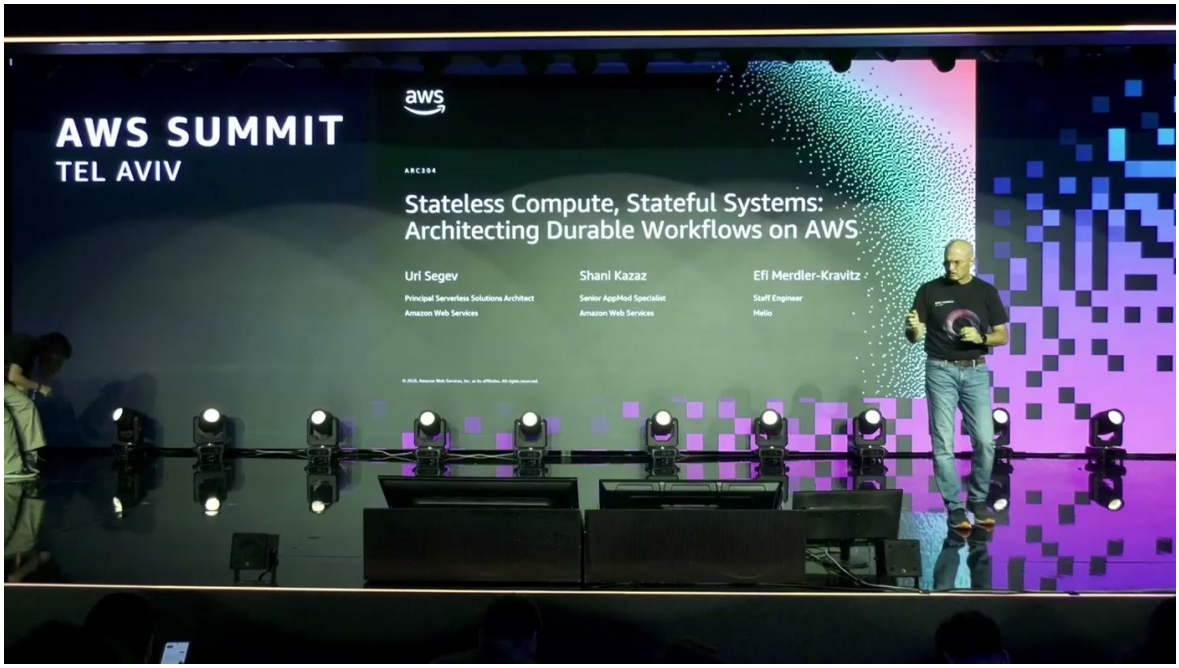
המסמך מסכם את הסשן ARC304 מתוך AWS Summit Tel Aviv 2026. הסשן עוסק במתח שבין קומפיוט סרברלס, שהוא חסר־מצב ומוגבל בזמן, לבין תהליכים עסקיים ותהליכים אג'נטיים שרצים דקות, ימים או שבועות ונדרשים לשרוד תקלות. הוא בנוי בשלושה חלקים: תיאוריה של אורקסטרציה מול כוריאוגרפיה, אבני הבניין והתבניות של AWS Lambda Durable Functions, וסיפור מימוש מהשטח בחברת Melio.

המסמך נבנה מתמלול אוטומטי של ההקלטה ומהסליידים שחולצו מהווידאו עצמו. חותמות הזמן בגוף המסמך מפנות לנקודה המדויקת בהקלטה.

איך לקרוא את זה

- **פרק 1 — תקציר מנהלים:** חמש דקות קריאה. מה הוצג ומה המסקנה המרכזית.
- **פרקים 2–4 — הרקע:** למה durability, למה מולטי־אג'נט שובר את הדטרמיניזם, ומהי בעיית הקואורדינציה.
- **פרקים 5–6 — התיאוריה והכלי:** אורקסטרציה מול כוריאוגרפיה, ומה נותן Lambda Durable Functions.
- **פרקים 7–8 — הפרקטיקה:** שש אבני הבניין ושלוש התבניות שנבנו מהן, עם הקוד שהוצג על המסך.
- **פרק 9 — מהשטח:** איך Melio בנתה מערכת השכרת סביבות AWS לאג'נטים מעל Durable Functions.
- **פרקים 10–11:** מה לוקחים מכאן, ומונחון.

> **הערת מקור** המסמך הופק מתמלול אוטומטי של ההקלטה. התמלול מדויק בקטעים המקצועיים, אך מונחים לועזיים ושמות מוצרים מופיעים בו בשיבוש פונטי (למשל "דורבל פנקשן" = Durable Functions, "סטפ" = Step) — בגוף המסמך הם נורמלו לכתיב האנגלי המקובל. הציטוטים מובאים כלשונם מהתמלול ואומתו מול חותמת הזמן המצוינת. שמות הדוברים ותפקידיהם נלקחו מסלייד הפתיחה, לא מהתמלול.



סלייד הפתיחה: שלושת הדוברים ותפקידיהם — שניים מ-AWS ואחד מ-Melio

1. תקציר מנהלים

הסשן פותח בהצהרה על המתח שהוא בא לפתור: "המתח היסודי שקיים בין הקומפיוט שלנו, שהוא סטייטלס, הוא סרברלס הרבה פעמים, לבין האפליקציות שלנו, הוורקפלווס, שצריכים לרוץ לאורך זמן, שהם צריכים להתמודד עם תקלות" [0:00]. משם הוא נע בקו ישר: למה צריך עמידות, איזו מתודולוגיה לבחור, אילו אבני בניין קיימות, ומה קורה כשמישהו באמת בונה עם זה מערכת בפרודקשן.

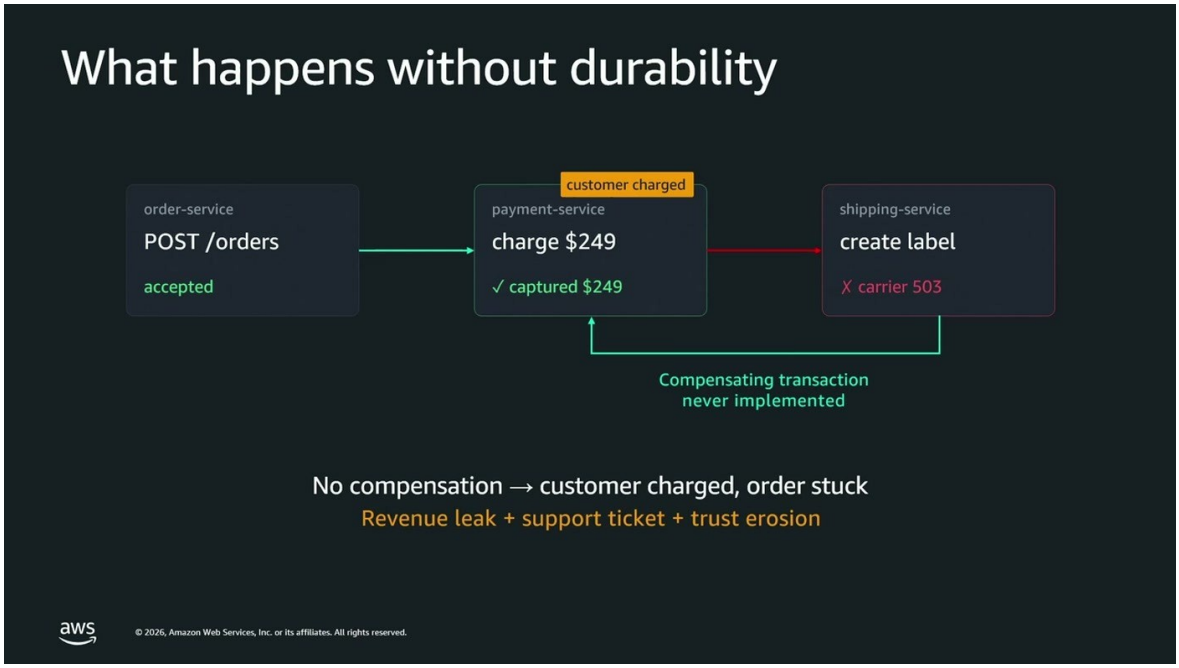
- **מולטי־אג'נט הוא ארכיטקטורת מיקרו־שירותים, על כל המחיר שלה.** הפיצול לאג'נטים מתמחים נותן את אותם יתרונות — אונרשיפ, דיפלוימנט נפרד, fault isolation — ומוסיף שניים ייחודיים: מודל מתאים לכל משימה, וריבוי חלונות הקשר קטנים במקום אחד גדול.
- **אג'נטים אינם דטרמיניסטיים, ולכן אינם מתאימים לכל workflow.** "בכל המקרים האלה או ברובם לפחות מי שמחליט מה עושים הלאה זה האג'נט" [4:49]. כשנדרש סדר צעדים מחייב — הדוגמה שניתנה היא תהליך אישור הלוואה בבנק — צריך אורקסטרטור חיצוני ולא להסתמך על המודל.
- **הבחירה בין אורקסטרציה לכוריאוגרפיה היא בחירה בין נראות לבין ניתוק.** אורקסטרציה נותנת global state, דיבוג ואודיטינג פשוטים, במחיר coupling ונקודת כשל מרכזית; כוריאוגרפיה הפוכה בדיוק. ההמלצה בפועל: אורקסטרציה בתוך דומיין, כוריאוגרפיה בין דומיינים.
- **Lambda Durable Functions מביא עמידות לקוד Lambda רגיל.** צ'קפוינט אחרי כל צעד, replay מאותה נקודה אחרי קריסה או השעיה, והכל דרך הרחבה ל-Context Object הקיים. בזמן המתנה — "שום דבר לא רץ, אף אחד לא משלם שום דבר" [12:28].
- **שש אבני בניין מספיקות כמעט לכל תבנית.** Task, Retry, Branching, Wait, Parallel ו-Map. מהן נבנו על הבמה שלוש תבניות מלאות: Scatter-gather, Saga ו-Human in the loop.
- **Melio בחרה ב-Durable Functions על פני Step Functions בגלל הקוד.** שלושה נימוקים: אין DSL נפרד מהלוגיקה, אפשר להריץ יוניט־טסטים אמיתיים מקומית, ואפשר לתפור שירותים שכבר כתובים בלי לכתוב אותם מחדש.

2. למה בכלל צריך durability

הפתיחה לא הגדירה durability אלא הראתה את היעדרו. שתי דוגמאות, אחת אג'נטית ואחת עסקית קלאסית.

הראשונה: אג'נט מחקר שרץ שמונה־עשרה דקות, עובר על מסמכים ומנתח אותם, וקורס שתי דקות לפני הסוף. "אם אין לנו durability, צריך להתחיל הכל מחדש. בזבזנו טוקנים, בזבזנו זמן, בזבזנו כסף, זה חבל" [0:00].

השנייה מוצגת על הסלייד: מערכת הזמנות שחייבה לקוח ב-249 דולר ואז נכשלה ביצירת המשלוח. "יש לנו בעצם טרנזקציה חלקית שבה היוזר חויב אבל הוא לא יקבל את המשלוח שלו" [1:42]. ללא מנגנוני פיצוי, התוצאה היא לקוח לא מרוצה, פניות לשירות, ופגיעה במוניטין.

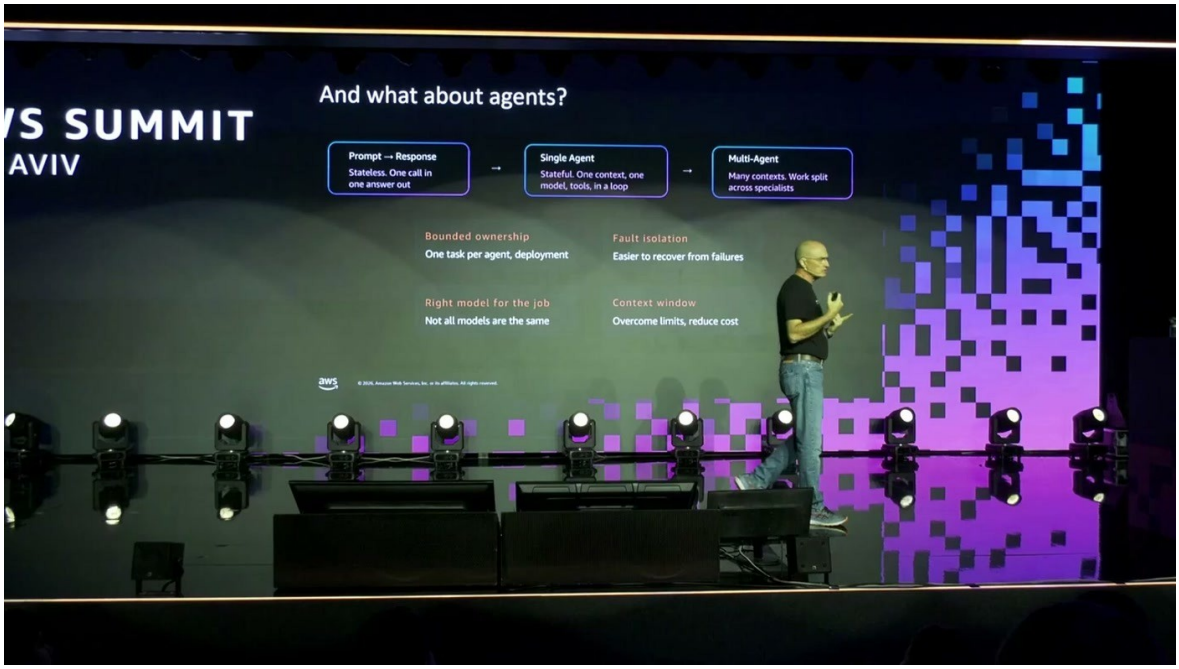


הטרנזקציה החלקית: חיוב הצליח, יצירת המשלוח החזירה שגיאה 503, ופעולת הפיצוי מעולם לא מומשה

3. מצ'אט לאג'נט בודד למולטי־אג'נט

כדי להסביר למה durability רלוונטי במיוחד לעולם האג'נטי, הוצגה אבולוציה בת שלושה שלבים. בהתחלה צ'אט — שולחים שאלה, מקבלים תשובה, אין ביצוע פעולות. אחר כך אג'נט בודד: מודל, סט כלים, system prompt ו-skills, והוא מחליט בעצמו מתי להפעיל איזה כלי. השלב הבא הוא פיצול לאג'נטים קטנים ומתמחים.

הנימוק לפיצול הוא אותו נימוק שהוביל מהמונולית למיקרו־שירותים: "בדיוק כמו שאנחנו שוברים את המונוליד שלנו למייקרוסרוויסס, גם פה יש לזה אותם יתרונות" [1:42]. צוות אחד אחראי על אג'נט אחד, "האג'נט עושה דבר אחד ועושה אותו יותר טוב" [3:17], אפשר לעשות דיפלוימנט לכל אג'נט בנפרד, ואפשר לבודד תקלות ולהריץ מחדש רק את האג'נט שנכשל.

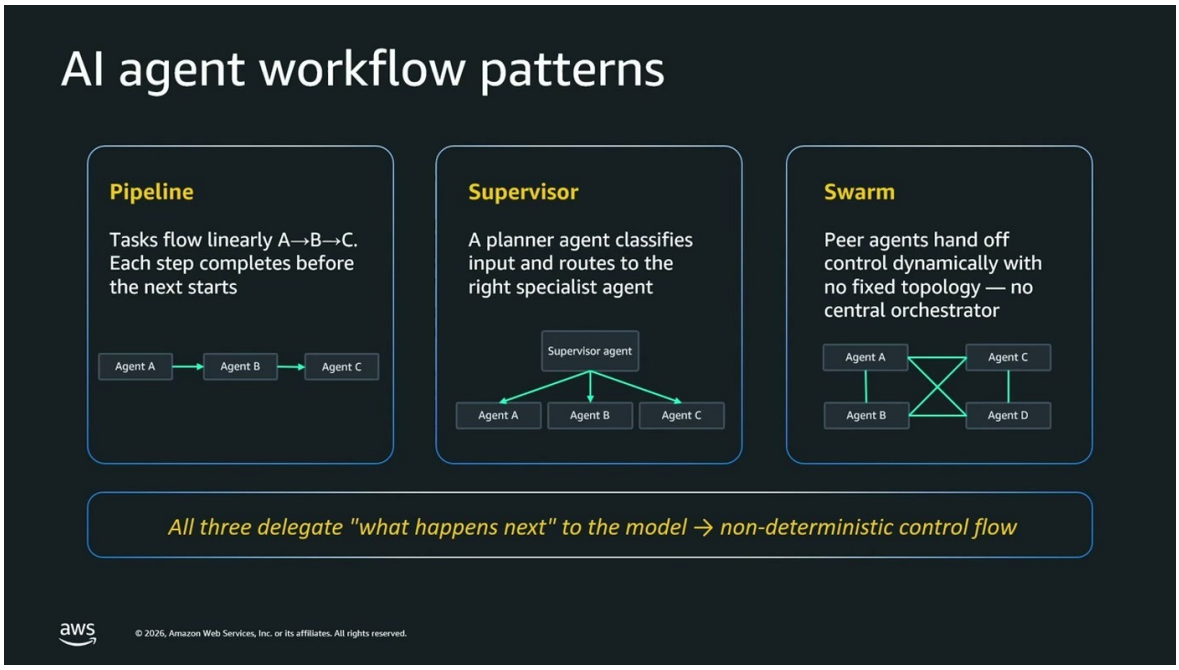


שלושת השלבים ומתחתם ארבעת הנימוקים לפיצול: אונרשיפ תחום, בידוד תקלות, המודל הנכון למשימה, וחלון הקשר

שני נימוקים נוספים ייחודיים לאג'נטים. הראשון הוא עלות: אג'נט אחד גדול מחייב לבחור מודל אחד, והוא חייב להיות החזק ביותר שנדרש לכל פעולה — כלומר משלמים מחיר של מודל כבד גם על קלסיפיקציה פשוטה. בפיצול, "כל אייג'נט יכול לעשות להשתמש במודל שמתאים לו" [3:17]. השני הוא חלון ההקשר: הפיצול מחליף חלון אחד גדול בכמה חלונות קטנים, ומקל על המגבלה.

שלוש שיטות תיאום — וכולן לא דטרמיניסטיות

הוצגו שלוש שיטות לתיאום בין אג'נטים: Pipeline, שבו אג'נט אחד מפעיל את הבא אחריו בסדר קבוע; Supervisor, שבו אג'נט מתכנן מקבל את הקלט ומחליט אילו אג'נטים להפעיל ובאיזה סדר; ו-Swarm, שבו כל אג'נט מחליט בעצמו מתי להעביר את השרביט, "ואין לנו שום שליטה לדבר הזה" [4:49].



שלוש התבניות, ומתחת להן המשפט שמחבר אותן: כולן מוסרות למודל את ההחלטה מה קורה הלאה

הנקודה שלשמה הן הוצגו איננה ההבדל ביניהן אלא המשותף להן: "בכל המקרים האלה או ברובם לפחות מי שמחליט מה עושים הלאה זה האג'נט, אנחנו בעצם אין לנו דטרמיניזם" [4:49]. ברוב המקרים המודל יחליט נכון אם ניתן לו system prompt ו-skills טובים — אבל יש יישומים שבהם "ברוב המקרים" אינו מספיק.

> **Uri Segev, [4:49]–[6:19] —** אם אני עושה איזו אפליקציה להלוואות בבנק ויש לי שם צעדים מסוימים שאני צריך לעשות... אני צריך קודם כל לבדוק את התפסים, אחר כך אני צריך לבדוק את היסטוריית האשראי של הלקוח, ואז בסוף לשלוח את זה לאישור המנהל. הכל עובד מצוין, אבל אם פעם אחת האיג'נט לא יחליט לבקש מהמנהל אישור, זה יכול עלות לנו הרבה כסף, אולי אנחנו נעבור על החוק, יש רגולציות.

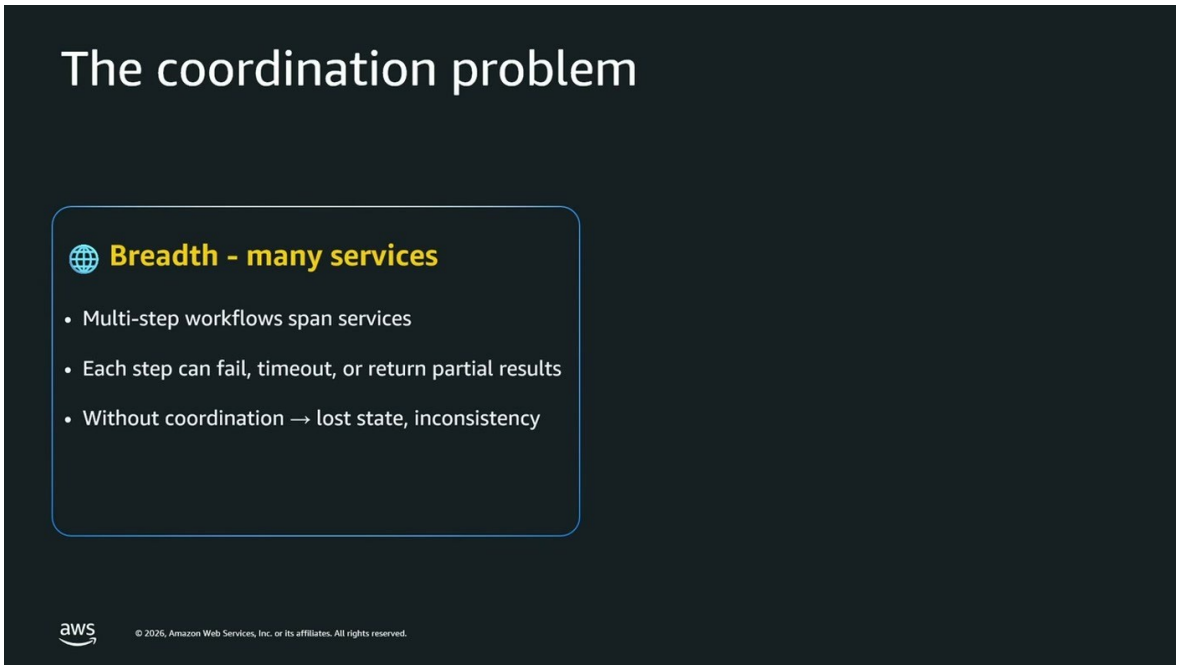
המסקנה המפורשת: "ולכן במקרים האלה, כשאנחנו צריכים וורקפלו שהוא דטרמיניסטי, אנחנו חייבים לעבור לכלי אורכסטרציה חיצוני, לא להצטמך על איג'נטים שיעשו את העבודה" [6:19].

4. בעיית הקואורדינציה: שני צירים

מעבר לדטרמיניזם, הוצגה בעיית הקואורדינציה כבעיה דו־צירית.

- **ציר הרוחב — ריבוי שירותים.** כלי האורקסטרציה מתאמים הרבה שירותים שונים, "השירותים האלה יכולים להיות, כל אחד מהם יכול להיכשל, יכול להיות טיימאוט, לחזיר תשובות החלקיות" [6:19]. בלי תיאום — אובדן מצב וחוסר עקביות.
- **ציר הזמן — תהליכים ארוכים.** "אנחנו יודעים שאיג'נטים לא מסיימים תוך שנייה את העבודה שלהם, יכול לקחת להם שבועות לפעמים חודשים" [6:19].

הדוגמה שמחברת בין השניים היא אנושית ולא טכנית: "אם אני מבקש אישור ממנהל יכול להיות שהמנהל יצא לחופש, בכלל אני לא יודע מתי אני אקבל את האישור הזה, ולכן אני צריך להיות מסוגל להתמודד עם דברים שלוקחים הרבה זמן" [6:19]. זו בדיוק המגבלה שהופכת את Lambda הקלאסית, עם תקרת ה-15 דקות, ללא מתאימה למשימה — וזה הפער שהחלק השני של הסשן בא לסגור.



ציר הרוחב כפי שהוצג: workflows רב־שלביים שחוצים שירותים, כל צעד עלול להיכשל, ובלי תיאום מאבדים מצב

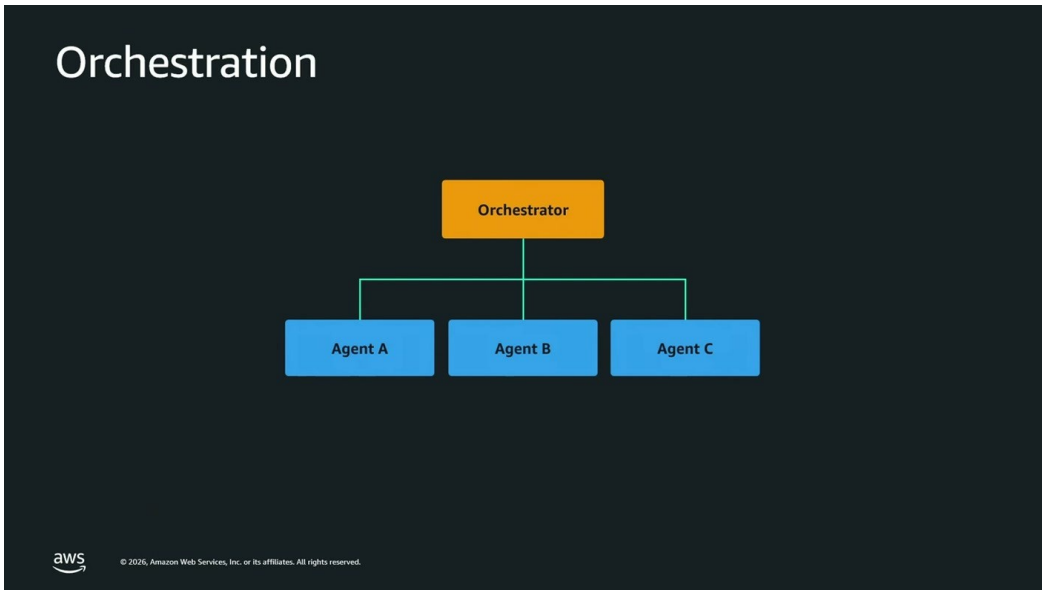
5. אורקסטרציה מול כוריאוגרפיה

החלק השני של הסשן, שהוצג על ידי ה-AppMod Specialist, פתח בשתי המתודולוגיות המרכזיות לבניית workflows.

אורקסטרציה

"אורקסטרציה זה מתודולוגיה שבה יש לי קומפוננטה אחת שמנהלת את כל הוורקפלוו שלי. אני ממש כותבת את הכל בתוך קוד, אני מגדירה מה אני בדיוק רוצה, מה הסטפים השונים, מה הדינמיקה ביניהם, מה הרטריז מקניזמס, הכל במקום אחד" [7:53].

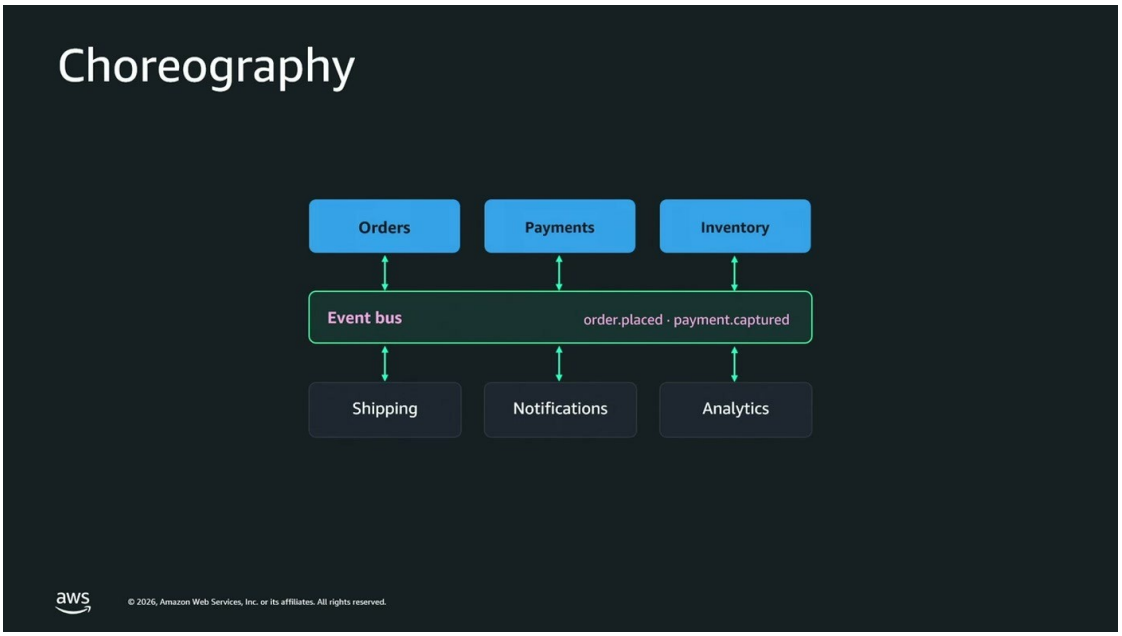
היתרון: יש ישות אחת שיודעת הכל, ולכן קל להבין את ה-global state ולעשות דיבוג ואודיטינג. החיסרון: coupling בין הרכיבים, ו"אם אורקסטרטור נופל, זה אומר שכל ה-workflow שלי נופל" [7:53].



אורקסטרטור מרכזי אחד מפעיל שלושה אג'נטים — כל הידע על הזרימה יושב במקום אחד

כוריאוגרפיה

בכוריאוגרפיה אין רכיב מנהל. ה-workflow מתנהל דרך ההודעות שזורמות בין השירותים: כל שירות עושה subscription ל-Event Bus, מאזין להודעות שמעניינות אותו, מבצע פעולה ושולח הודעה בחזרה. היתרון הוא ניתוק — "אם השירותים השונים שלי קמים והולכים, אז הרבה יותר קל לנהל את זה, כי אף אחד לא מכיר אף אחד בעצם" [9:24]. החיסרון הוא הצד השני של אותו מטבע: "לשמור את הגלובל סטייט, לעשות דיבאגינג ואודיטינג, זה קצת יותר מאתגר, כי אף אחד לא באמת בנה את ה-workflow הזה" [9:24].



Orders, Payments ו-Inventory מפרסמים ל-Event Bus; Shipping, Notifications ו-Analytics מגיבים — הזרימה מתקיימת רק כסכום התגובות

מתי בוחרים במה

בוחרים אורקסטרציה כאשר	בוחרים כוריאוגרפיה כאשר
הצעדים ברורים ותלויים זה בזה	התהליכים פשוטים יותר
יש חובת אודיטינג	הצרכנים קמים ונעלמים
נדרש Human approval / Human in the loop	הניתוק בין הרכיבים הוא הדרישה המרכזית
כשל של ה-workflow אמור להכשיל את התהליך כולו	

והמסקנה: לא באמת צריך לבחור. "אנחנו הרבה פעמים נראה שילוב של שתי המתודולוגיות האלה בארכיטקטורה אחת, אנחנו נראה אורקסטרציה בתוך דומיין, ובין דומיינים אנחנו יכולים להשתמש בחירורגרפיה, וככה בעצם להנות מכל העולמות" [9:24].

6. Lambda Durable Functions

להרצת אורקסטרציה על AWS הוצגו שלוש דרכים: self-managed — כלים כמו Temporal או Argo Workflows שמותקנים ומנוהלים עצמאית מעל EKS, ECS או EC2; מנוהל — Amazon Managed Workflows for Apache Airflow (MWAA); וסרברלס מלא — Step Functions, EventBridge ו-Lambda.

לצורך הדגמת התבניות נבחר שירות אחד: "אז אנחנו בחרנו בלמדה דורבל פנקשן, זה פיצ'ר חדש שיצא ללמדה ממש לא מזמן, והוא מאפשר לי לבנות וורגפלווס דורבל על גבי השירות של למדה" [10:55]. הודגש שהתבניות עצמן ניתנות למימוש בכל אחד מהכלים.

הרקע: Lambda נבנתה כשירות Function as a Service — פונקציה שעושה דבר אחד או כמה צעדים פשוטים, רצה עד 15 דקות, "ואם באמצע אינבוקציה ה-Lambda שלי קרסה, אני אצטרך להתחיל את האינבוקציה מחדש, זאת אומרת שהיא לא דורבילית" [10:55].

שלושת המרכיבים של durable execution: שמירת התקדמות, המשך ריצה אחרי השעיה או הפרעה, ו-SDK שמרחיב את ה-event handler

המנגנון: "אני יכולה לשמור צ'קפוינטים אחרי כל צעד ב-Workflow שלי, אני שומרת את הריזולט במקום אחר לגמרי, מה שמאפשר לי לעשות ריפלי — אם הפונקציה שלי קרסה או שבחרתי להשעות אותה, אני יכולה לחזור בדיוק לאותה נקודה" [12:28].

הדגמה: שני צעדים והמתנה של עשר שניות

האפליקציה שהודגמה היא הפשוטה ביותר האפשרית: צעד ראשון, המתנה של עשר שניות, צעד שני. הריצה מתחילה, הצעד הראשון עובר בהצלחה והתוצאה נשמרת חיצונית. בנקודת ההמתנה האינבוקציה נכבית — "שום דבר לא רץ, אף אחד לא משלם שום דבר" [12:28] — ושירות Lambda מעיר אינבוקציה חדשה כעבור עשר שניות. באינבוקציה השנייה, כל צעד שכבר בוצע פשוט נשלף מהצ'קפוינט במקום להתבצע שוב: "אני מגיעה לצעד הראשון, אני רואה שכבר רצתי, אז אני לא עושה שום דבר, אני פשוט שולפת את התוצאות" [12:28]. רק הצעד השני מבוצע בפועל.

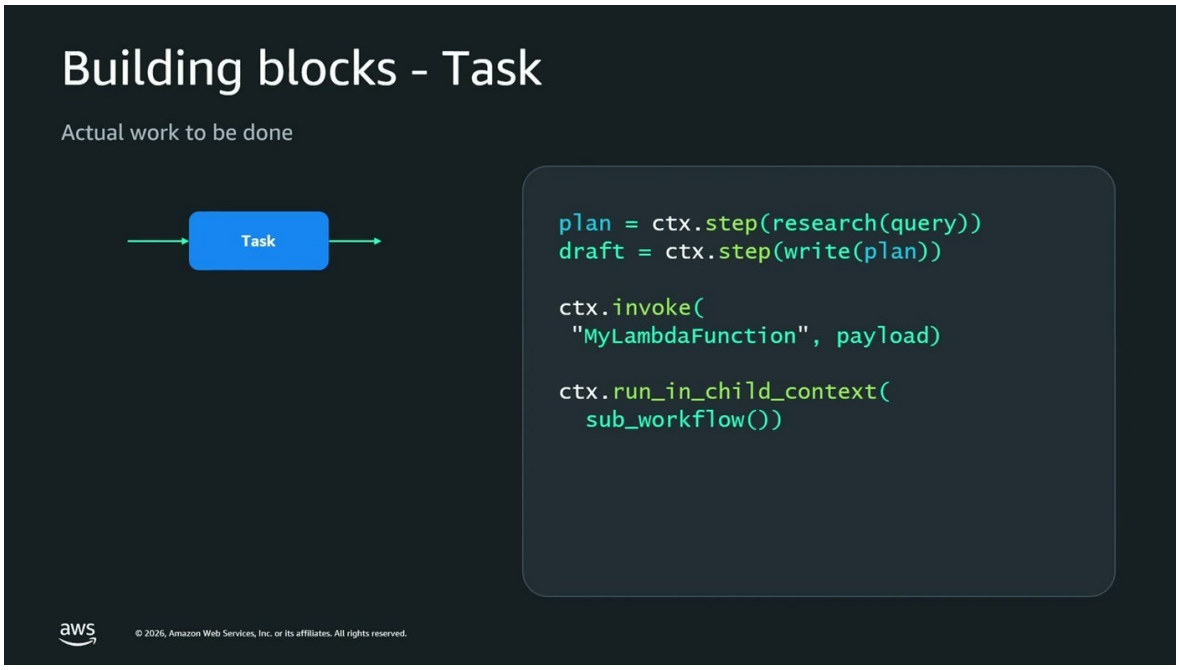
המשמעות התפעולית זמן ההמתנה אינו זמן חישוב. זה נכון להמתנה מפורשת, להמתנה בין ניסיונות retry, ולהמתנה ל-condition או ל-callback חיצוני — וזה מה שהופך תהליך שנמשך ימים לכדאי כלכלית מעל Lambda.

7. שש אבני הבניין

לפני התבניות הוצגו אבני הלגו שמהן הן נבנות. כולן נחשפות דרך ה-Context Object של Lambda: "לפונקציות Lambda תמיד מקבל את שני הפרמטרים... יש את ה-Event Object שאומר למה היא הופעלה, ויש את ה-Context Object, שעד היום לא השתמשנו בו כמעט. אבל הרחבנו אותו עכשיו לתמוך בדורבל אקזקיושנז" [14:00].

1. Task — העבודה עצמה

"האורקסטרטור עצמו לא עושה עבודה. הוא מתאם דברים אחרים" [14:00]. ה-Task הוא מה שהוא מתאם. הקריאה ל-ctx.step מפעילה מתודה בקוד, מחזירה את תוצאתה ומבצעת את הצ'קפוינטינג אוטומטית: "ברגע שעטפתי את הקוד שלי בתוך סטפ, זה יקרה לבד" [15:32].



שלוש הדרכים לבצע עבודה: step על מתודה מקומית, invoke לפונקציית Lambda אחרת, ו-run_in_child_context לתת־workflow

ל-ctx.invoke יש נימוק כלכלי מפורש. עד היום לא הומלץ להפעיל Lambda מתוך Lambda באופן סינכרוני, "כי אם אני מפעיל פונקציית lambda בצורה סינכרונית מתוך פונקציית lambda אחרת, אז אני משלם על שתי הפונקציות במקביל. אנחנו לא אוהבים לבזבז כסף" [15:32]. עם invoke הדורבילי, הפונקציה הקוראת יורדת ברגע ההפעלה, וכשהשנייה מסתיימת שירות Lambda מעיר אותה בנקודה שאחרי ה-invoke.

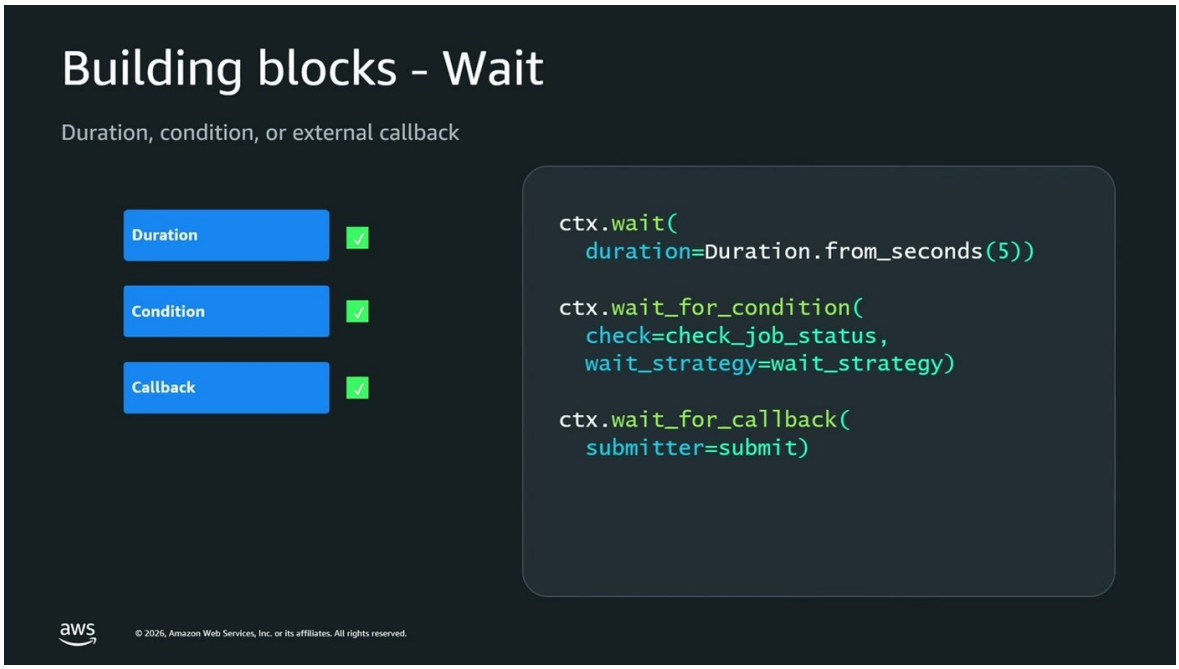
2. Retry

מעבירים retry strategy ל-step: מספר ניסיונות מרבי, זמן המתנה בין ניסיונות, שימוש ב-jitter ופרמטרים נוספים. "זה יקרה אוטומטית בשבילי, אני לא צריך לעשות שום דבר נוסף חוץ מאשר להגדיר את זה" [15:32]. ובזמן ההמתנה בין ניסיון לניסיון — "הפונקציה לא רצה" [17:05].

3. Branching

כאן אין בכלל מנגנון ייעודי, וזו הנקודה: "לא צריך שום דבר מיוחד בדורבל... אני פשוט עושה if, אני עושה case, switch, whatever" [17:05]. המסלול יכול להיות תלוי בקלט מהמשתמש או בפלט של צעד קודם, והכל נכתב בשפת התכנות עצמה.

4. Wait — שלושה סוגים



המתנה לפרק זמן, המתנה עד שתנאי מתקיים, והמתנה ל-callback חיצוני — שלושתן בלי חישוב פעיל

- **המתנה למשך זמן.** ctx.wait עם duration. כשהזמן עובר, הריצה מתעוררת ומגיעה לאותה נקודה.
- **המתנה לתנאי.** ctx.wait_for_condition מקבל שתי מתודות: אחת שבודקת את התנאי (למשל פנייה לבסיס נתונים או בדיקת סטטוס job), ו-wait_strategy שקובעת כמה זמן לישון בין בדיקה לבדיקה — "אני לא רוצה לעשות ביזי לופ כזה שהוא כל הזמן בודק" [18:35].
- **המתנה ל-callback.** ctx.wait_for_callback מקבל פונקציית submitter ששולחת טוקן למערכת חיצונית; כשהמערכת החיצונית קוראת בחזרה עם הטוקן, הריצה מתעוררת.

5. Map · 6. Parallel

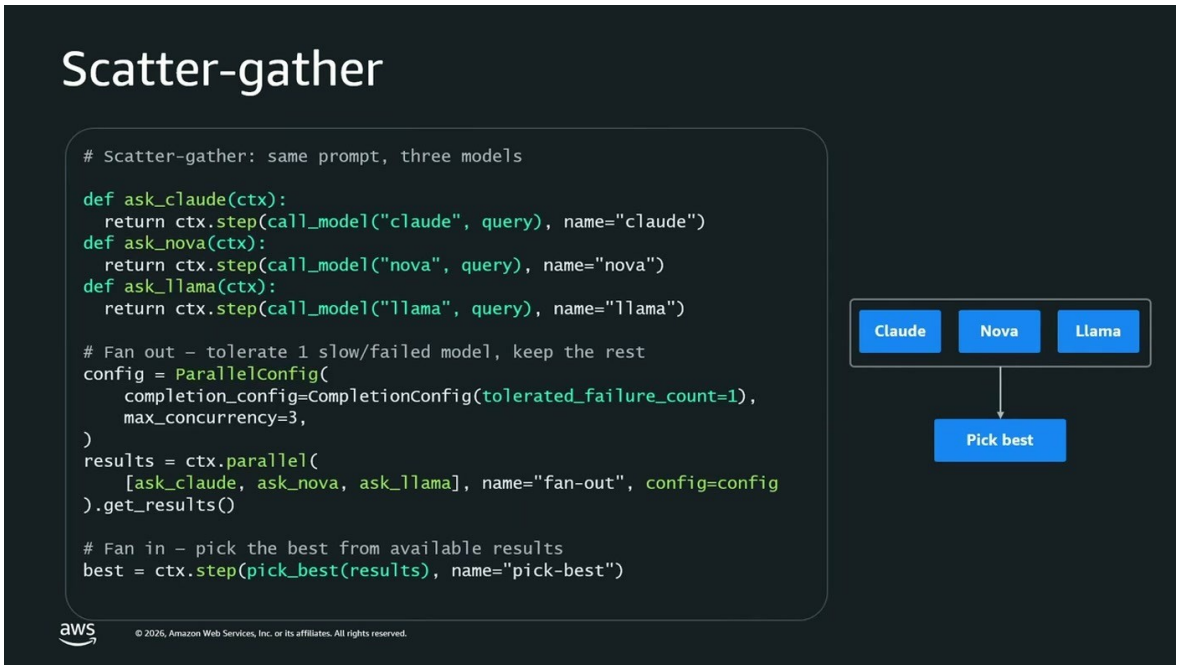
ctx.parallel מקבל מערך של צעדים שירוצו במקביל, ולצדו קונפיגורציה — למשל כמה צעדים רשאים להיכשל מבלי שהפעולה כולה תיחשב ככישלון. ctx.map מקבל מערך ערכים ופונקציה אחת שתרוץ על כל אחד מהם, גם כאן עם קונפיגורציה של סף שגיאות.

שניהם נראים דומה, וההבחנה חדה: "ההבדל העיקרי זה שפרללל אני בעצם עושה דברים שונים על אותו אינפוט בדרך כלל, ובמאפ אני עושה את אותו דבר על ערכים שונים, אינפוטים שונים" [20:06].

8. שלוש תבניות

8.1 Scatter-gather (Fan-out / Fan-in)

הרצת כמה משימות במקביל ואיגוד התוצאות, עם הגדרה של כמה כישלונות מותרים כדי שה-workflow עדיין יצליח. הדוגמה שנבחרה מוכרת לכל מי שעובד עם מודלים: "אנחנו רוצים להריץ את אותו query עם כמה מודלים שונים. קלאסי" [20:06].

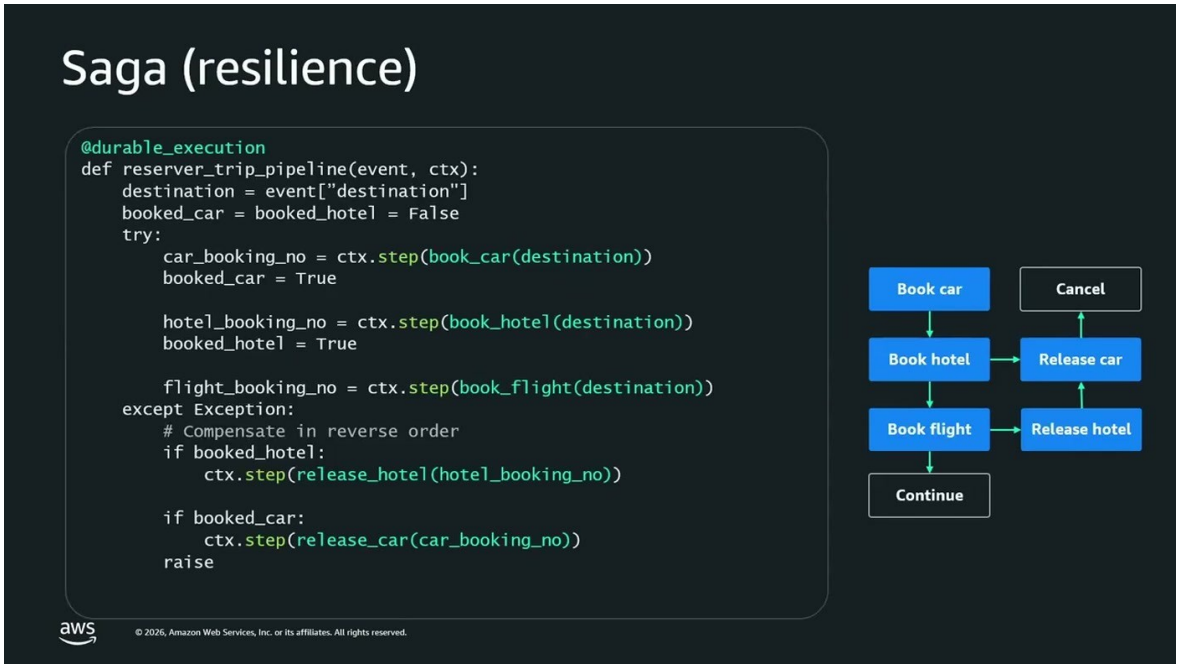


אותו prompt לשלושה מודלים במקביל עם סובלנות לכישלון אחד, ואז בחירת התוצאה הטובה ביותר

בקוד שעל המסך: שלוש פונקציות עטופות ב-ctx.step, ParallelConfig עם tolerated_failure_count=1 ו-max_concurrency=3, קריאה ל-ctx.parallel ולבסוף step של pick_best על התוצאות שהגיעו. הוער שאותה תבנית ניתנת למימוש גם ב-Map, כששמות המודלים הם ערכי המערך [21:46].

Saga 8.2 — טרנזקציות מפצות

במונולית עם בסיס נתונים רלציוני אחד יש BEGIN, COMMIT ו-ROLLBACK. במיקרו־שירותים לכל שירות בסיס נתונים משלו, לעיתים לא רלציוני, ו"אין לי אפשרות לעשות טרנזקציה מבוזרת כזאת" [21:46]. התחליף הוא Saga: "כל microservice צריך לתת לי שתי פעולות: אחת פעולה שעושה את העבודה, ופעולה אחרת שעושה את האנדו של הפעולה הזאת" [21:46]. בכישלון רצים אחורה ומבצעים undo לצעדים שהצליחו, בסדר הפוך.



הזמנת טיול: רכב, מלון וטיסה ב-try, ובלוק except שמשחרר בסדר הפוך רק את מה שהוזמן בפועל

הפרט שהודגש בקוד הוא דגלי המצב. אחרי כל הזמנה מוצלחת נכתבת שורה כמו booked_car = True — "זה אומר שהצלחתי להזמין את הרכב" [23:17] — ובלוק ה-except משתמש בדגלים כדי לדעת מה בכלל צריך לבטל: "אם

הצלחתי להזמין את המלון, תשחרר את המלון. אם הצלחתי להזמין את הרכב, תשחררו את הרכב" [23:17]. כשל ב-T2 מחזיר ישירות ל-C1 בלי להגיע ל-T3.

Human in the loop 8.3

התבנית השלישית, שהוצגה כבלתי נמנעת בעולם האג'נטי: שולחים בקשה וממתינים להחלטת אדם. "ה-Workflow עוצר, שום דבר לא קורה עד שאין לי את האישור או את הדחייה של הבקשה מהבן אדם" [23:17], וההמשך תלוי בהחלטה.

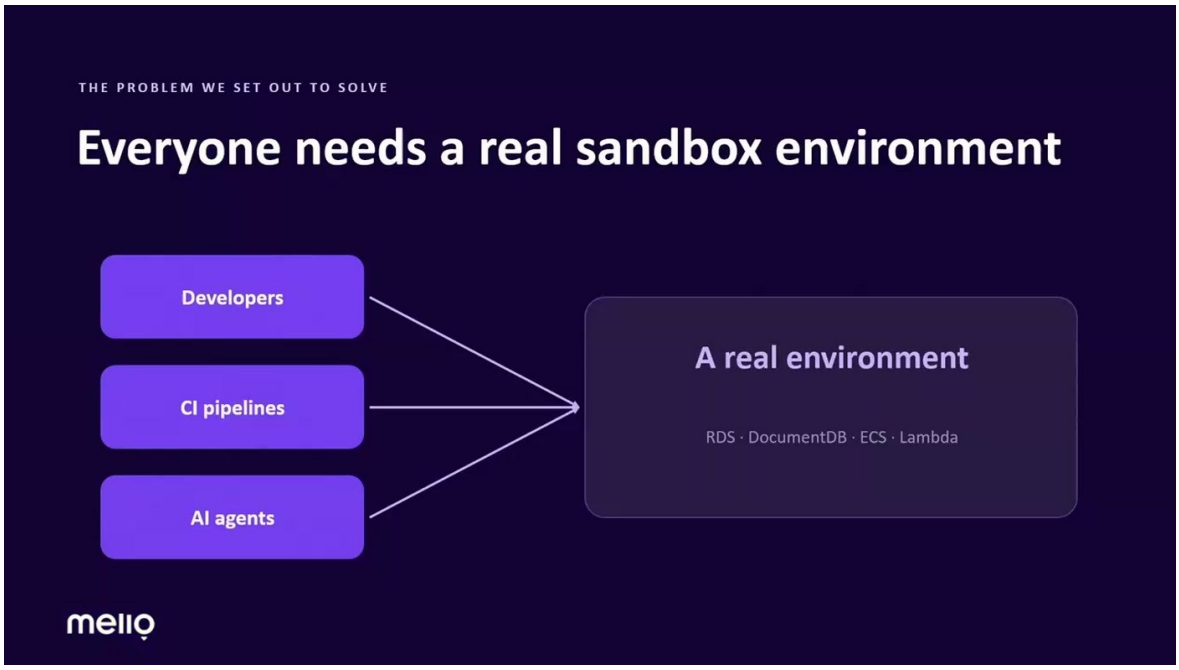
הדוגמה: "הדוגמה הקלאסית ביותר, לשלוח מייל" [24:52] — אג'נט שקיבל יותר מדי הרשאות ושלח מייל בטעות. המימוש מייצר draft, ואז קורא לפונקציית submit שמפעילה את wait_for_callback; ה-workflow ממתין עד שהאדם מכריע, ורק אז המייל נשלח או לא.

9. מהשטח: Durable Functions בפרודקשן ב-Melio

החלק השלישי הועבר על ידי Staff Engineer מ-Melio — "מיליו זה חברת בי-טו-בי שעוזרת לעסקים קטנים להעביר כספים אחד לשני" [26:24]. הוא הציג את השימוש שלהם ב-Durable Functions ואת הנימוקים לבחירה.

הבעיה: סביבות סנדבוקס אמיתיות, בסקייל

"אני חושב שהיום כולם צריכים סביבות סנדבוקס כדי לבדוק את הקוד שלהם" [24:52] — אג'נטים, מערכות CI, וגם בני אדם. אפשר להריץ מקומית, אבל להרשאות, אינטגרציות ומגבלות AWS נדרשת סביבת AWS אמיתית. וכשיש סביבה אחת זה נוח, "אבל מה כש יש לנו מאות סיבות כאלו? כן, תדמיינו, יש לכם מאות אייג'נטים בארגון, יחד עם עשרות עד אלפי ריצות של CI בארגון" [26:24].



שלושת סוגי הצרכנים — מפתחים, פייפליינים של CI ואג'נטים — מבקשים כולם את אותו דבר: סביבה אמיתית

שלוש בעיות הוצגו: עלות — רכיבים כמו RDS עולים כסף גם כשלא משתמשים בהם; ניקוי — הסביבות עוברות ממפתח לאג'נט ל-CI וצריך להחזירן למצב נקי בסקייל גבוה; ושכפול — CloudFormation מצוין, אבל "יש המון ידע בעל פה, המון קונפיגורציות שלאו דווקא רשומות בתוך ה-CloudFormation" [26:24].

המערכת

הדרישות שהוגדרו לסביבה: אקסקלוסיבית לצרכן שלה, כי "אנחנו לא רוצים שהאג'נטים ידרכו אחד לשני על הרגליים בזמן העבודה" [27:58]; מוגבלת בזמן, גם מסיבות עלות וגם כי "לאג'נטים לפעמים יש נטייה להתברבר" [27:58] ותיחום בזמן עוזר לאפס אותן; וכשאינה בשימוש — שתעלה את המינימום האפשרי.

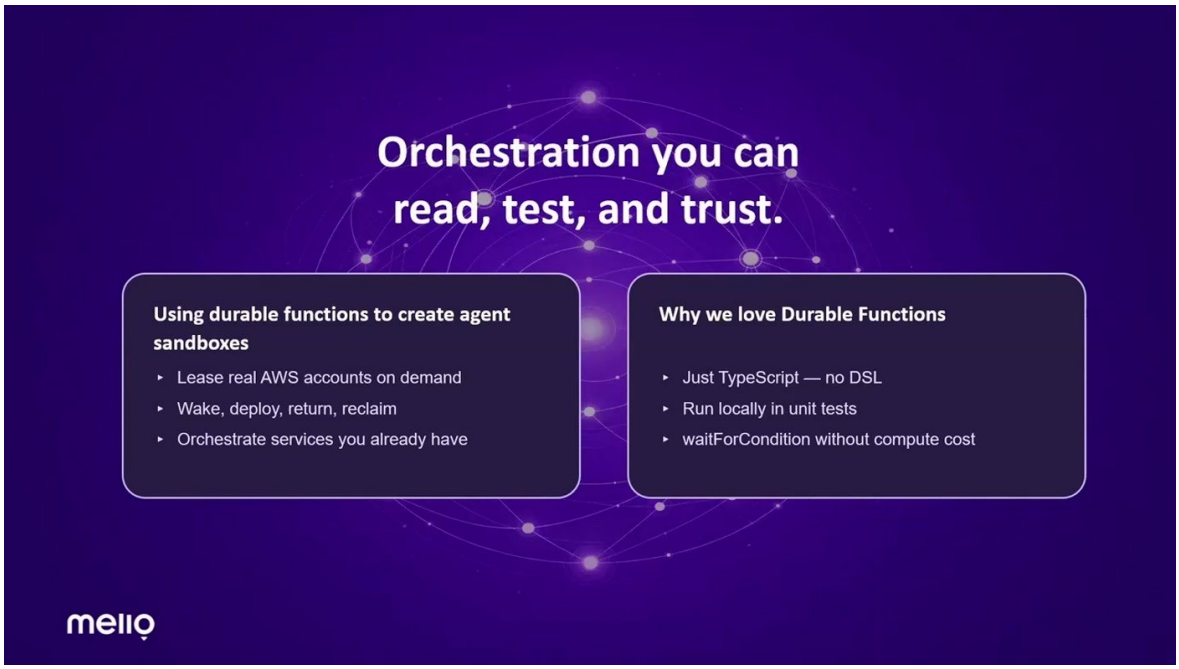
הזרימה: אג'נט פונה לשירות מרכזי שנקרא ה-orchestrator ומבקש sandbox. ה-orchestrator בוחר חשבון פנוי מתוך pool, מעיר אותו — "הסביבות האלו במצב רגיל, הן במצב של hibernation" [29:30], כלומר שירותים יקרים כמו RDS מורדים ומועלים בחזרה לפי דרישה — ומחזיר credentials. כשהאג'נט מסיים או שהזמן נגמר, הסנדבוקס מוחזר, מנוקה (מחיקת קוד, החזרת stacks, איפוס נתונים), מוכנס חזרה ל-hibernation ומוחזר ל-pool.

שני שימושים קונקרטיים באבני הבניין

- **wait_for_condition במקום polling.** ניקוי הסביבה הוא תהליך אסינכרוני ארוך. "אופציה אחת היא פולינג... פולינג סטנדרטי בלמדה מאוד לא יעיל, גם כי למדה מוגבלת בזמן 15 דקות, וגם כמו שהוא הזכיר, זה עולה כסף" [31:20]. במקום זאת נשמר ערך ב-DynamoDB שמסמן שהניקוי הסתיים, ו"הלמדה הזאת ישנה, מתעוררת באמצעות המנגנון של הדיורבל פנקשן, בודקת אם הערך הזה הוא true, אם כן היא מתקדמת הלאה, אם לא היא נרדמת שוב עד הפעם הבאה" [31:20]. אותה תבנית משמשת גם להורדת הסביבה ל-hibernation.

- **Map להחזרת CloudFormation stacks.** אותו רצף פעולות על כמות משתנה של stacks. היתרון שהודגש הוא ה-retry הממוקד: בתקלה "כל המערכת לא צריכה לעבור שוב על כל הסטאקים ולעבור על כל התהליך הזה מחדש, אלא יכולה לחזור שוב ולטפל אך ורק בסטאק הבעייתי" [32:52].

למה Durable Functions ולא Step Functions



סיכום המימוש מצד Melio: מה בנו עם Durable Functions, ושלושת הנימוקים לבחירה

שלושה נימוקים הוצגו, וההשוואה ל-Step Functions נאמרה במפורש ובלי לזלזל בהם ("סטפ פנקשן, מדהימים").

- **הכל בקוד, בלי DSL.** "עצם העובדה שצריך לעשות הפרדה בין הקוד לבין ה-DSL, גם ברמה האנושית של מישהו בא וקורא את הקוד, זה מורכב" [32:52]. ההיגיון הזה הורחב גם לאג'נטים: הלוקליות — הסדר והקוד באותו מקום — מקלה גם עליהם.
- **יוניט־טסטים אמיתיים מקומית.** "יש SDK רשמי ל-Durable Function עבור יוניטסטים, גם לנוד, גם לפייטון" [34:27], עם אינטגרציה מובנית למוקים של קריאות ללמבדה. ריצות מהירות, ואפשר לבדוק שהזרימה שתוכננה עם כל התנאים בקוד באמת רצה כמתוכנן.
- **תפירה של קוד קיים.** "המון אלמנטים בתוך המערכת כבר היו לנו כתובים, המון סרוויסים, אנחנו צריכים בעצם לעשות אורכסטרציה של קוד" [34:27] — הקוד היה מוכן, נדרש רק לחבר אותו נכון.

10. מה לוקחים מכאן

- **התבניות אוניברסליות.** "הפאטרים האלה שלמדנו הם אוניברסליים, הם טובים גם לאגנטים וגם טובים למייקרוסרוויסס, ואפשר לשלב ביניהם כשצריך" [36:36]. מי שכבר מכיר Saga ו-Scatter-gather ממיקרו־שירותים לא לומד כאן עולם חדש, אלא מקבל שם חדש להרצה שלהן.
- **סט קטן של אבני בניין מספיק.** "ראינו שעם סט קטן של אבני בניין אפשר לבנות כמעט כל אורכסטרישן פאטרן שאנחנו רוצים" [36:43]. שש אבני הבניין כיסו את שלוש התבניות שהוצגו וגם את מערכת הפרודקשן של Melio.
- **הכלל המעשי: היכן שנדרש סדר מחייב — אורקסטרטור.** אג'נטים מצוינים בהחלטה מה לעשות הלאה, ולא מתאימים במקום שבו צעד חייב להתבצע תמיד. בתהליכים רגולטוריים זו אינה שאלה של העדפה.
- **המתנה היא רכיב ארכיטקטוני, לא תקלה.** המתנה לאישור אנושי, לתנאי או ל-callback הפכה למשהו שאפשר לתכנן סביבו בלי לשלם על זמן חישוב — וזה מה שמכשיר את Lambda לתהליכים ארוכים.
- **בחירת הכלי היא שאלה של חוויית פיתוח.** אצל Melio ההכרעה בין Step Functions ל-Durable Functions לא נפלה על יכולות אלא על קריאות הקוד, יכולת הבדיקה המקומית, והתאמה לאג'נטים שקוראים את הקוד.

נקודות להמשך

נושא	מה נאמר	מה לבדוק
בשלות Durable Functions	"פיצ'ר חדש שיצא ללמדה ממש לא מזמן" [10:55]	זמינות באזורים, מגבלות, ותנאי שימוש עדכניים בתיעוד AWS
משך ריצה מרבי	צוין בהקלטה כמשך ארוך מאוד ביחס ל-15 הדקות של Lambda; הערך המדויק נאמר בשיבוש	לאמת מול התיעוד הרשמי לפני תכנון תהליך ארוך
שפות נתמכות	SDK ליוניט־טסטים "גם לנוד, גם לפייטון" [34:27]; הקוד על הסליידים ב-Python, ו-Melio מזכירים TypeScript	לבדוק את מטריצת ה-runtime הרשמית
תבניות תיאום אג'נטים	Pipeline, Supervisor ו-Swarm הוזכרו אך לא נדונו לעומק — הוסבו לסשן העוקב באותו מסלול	לצפות בסשן העוקב להשלמת הנושא

11. מונחון

מונח	פירוש כפי שהוצג בסשן
Durability	היכולת של תהליך לשרוד קריסה או השעיה ולהמשיך מהנקודה שבה עצר, בלי לחזור להתחלה
Durable Execution	מודל ריצה שבו נשמר צ'קפוינט אחרי כל צעד ותוצאתו נשמרת חיצונית, כדי לאפשר replay
Checkpoint / Replay	שמירת ההתקדמות, ולאחר התעוררות — דילוג על צעדים שכבר בוצעו ושליפת תוצאתם מהזיכרון החיצוני
אורקסטרציה	רכיב מרכזי אחד שמנהל את כל ה-workflow, מגדיר צעדים, סדר ומנגנוני retry במקום אחד
כוריאוגרפיה	ניהול ה-workflow דרך הודעות ב-Event Bus, ללא רכיב מנהל; כל שירות מגיב למה שמעניין אותו
Task / Step	יחידת עבודה בפועל; ctx.step עוטף מתודה ומבצע צ'קפוינט אוטומטי
Parallel	הרצת צעדים שונים במקביל על אותו קלט, עם סף כישלונות מותר
Map	הרצת אותו צעד במקביל על ערכים שונים במערך
wait_for_condition	המתנה עד שתנאי חיצוני מתקיים, עם אסטרטגיית המתנה בין בדיקות ובלי חישוב פעיל
wait_for_callback	המתנה לקריאה חוזרת ממערכת חיצונית עם טוקן שנשלח אליה מראש
Scatter-gather / Fan-out Fan-in	פיזור משימות במקביל ואיגוד התוצאות, עם הגדרה של כמה כישלונות נסבלים
Saga / Compensating Transaction	תחליף לטרנזקציה מבוזרת: כל שירות מספק פעולה ופעולת ביטול, ובכישלון רצים אחורה
Human in the loop	עצירת ה-workflow עד להכרעה אנושית, והמשך מותנה בהחלטה
Hibernation (במימוש של Melio)	מצב שבו רכיבים יקרים בחשבון הסנדבוקס מורדים, ומועלים בחזרה רק כשהסביבה מושכרת