רזיליינסי בארכיטקטורות Serverless

AWS Summit Tel Aviv 2026 ,סיכום סשן — ARC303

Maya Morav Freiman, Senior Technical Account Manager, AWS | Itay Shafir, Senior Solutions Architect, AWS

Itay Waisman, Principal Engineer, Sola Security

10 בספטמבר 2026 | משך: כ-38 דקות | הופק מהקלטת הסשן

סיכום מאת קובי אלמוג

על המסמך הזה

המסמך מסכם את הסשן ARC303 מתוך AWS Summit Tel Aviv 2026. הוא נבנה מתמלול אוטומטי של ההקלטה המלאה ומהסליידים שחולצו מהווידאו עצמו, כך שאפשר לחזור לתוכן בלי לצפות שוב בשלושים ושמונה הדקות. חותמות הזמן בגוף המסמך מפנות לנקודה בהקלטה.

איך לקרוא את זה

- **פרק 1 — תקציר מנהלים:** חמש דקות קריאה. מה נאמר ומה שווה לקחת.
- **פרקים 2–3 — המסגרת:** מה זה רזיליינסי ב-serverless, ומה ההנחות שאנחנו מניחים בלי לשים לב.
- **פרקים 4–6 — השירותים:** API Gateway, שלושת מודלי ההפעלה של Lambda, ו-EventBridge.
- **פרק 7 — סיפור לקוח:** Sola Security — ארכיטקטורת ה-ETL, המספרים, והתקלה בפרודקשן.
- **פרקים 8–9 — תבניות ומה לוקחים:** אינטגרציות ישירות, DynamoDB Streams, Step Functions ו-Saga.

הערת מקור המסמך הופק מתמלול אוטומטי של ההקלטה. הסשן נאמר בעברית עם מינוח טכני באנגלית, והתמלול משבש חלק מהמונחים הלועזיים — הם נורמלו כאן למונח האנגלי המקורי. הציטוטים אומתו מול התמלול בחותמת הזמן המצוינת, ומספרים שמופיעים על הסליידים צוינו ככאלה.

1. תקציר מנהלים

שלושה דוברים, שלושה חלקים: Maya Morav Freiman מסגרה מה זה רזיליינסי כשהתשתית כבר לא באחריותכם; Itay Shafir עבר שירות-שירות על מנגנוני הרזיליינסי המובנים ב-API Gateway, Lambda ו-EventBridge; ו-Itay Waisman מ-Sola Security הראה איך זה נראה במערכת אמיתית, כולל תקלה שהפילה לו את הפייפליין.

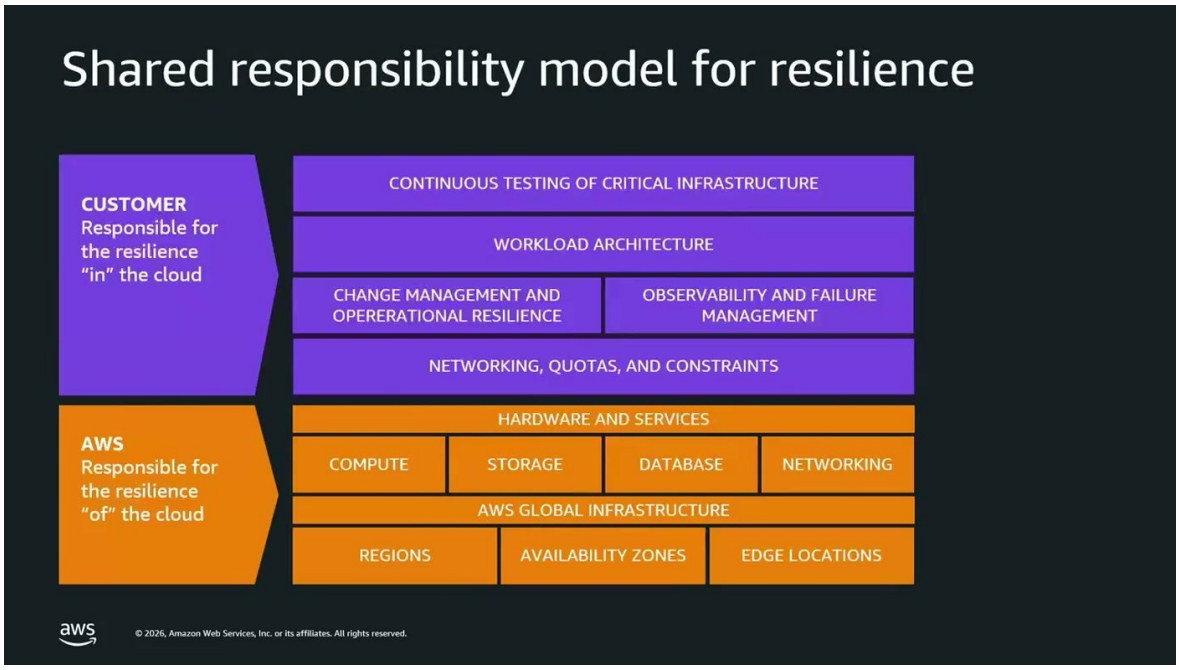
- **ב-serverless האחריות זזה, היא לא נעלמת.** AWS מקבלת את התשתית, ואתם נשארים עם ארכיטקטורת ה-workload. "רזיליאנסי אמיתי בהקשר הזה דורש בחירות ארכיטקטוניות מכוונות ברמת האפליקציה, לא רק ברמת התשתית" [3:05].
- **הקווים בדיאגרמה הם ההנחות שלא בדקתם.** תרגיל "שתי קופסאות וקו" פירק דיאגרמה מינימלית לשאלות שאין עליהן תשובה בציור: מי עושה polling למי, סינכרוני או אסינכרוני, אותו חשבון, אותו AZ, מה קורה ב-retry [4:35].
- **שלושת מודלי ההפעלה של Lambda מתנהגים אחרת לגמרי בשגיאה.** סינכרוני — אין retry אוטומטי ואין נקודת התאוששות; אסינכרוני — עד שני ניסיונות נוספים ואז DLQ; Event Source Mapping — התור שלכם, השליטה שלכם, ועיבוד ב-batch.
- **EventBridge נותן חלון התאוששות ארוך במיוחד.** לפי הסלייד: retry לאורך 24 שעות ועד 185 ניסיונות, עם exponential back-off ו-jitter, ו-DLQ עם ERROR_CODE ו-ERROR_MESSAGE לצוות ה-on-call [16:57].
- **אצל Sola, SQS+DLQ לפני כל רכיב compute הוא כלל ברזל.** "לפני כל רכיב קומפיוט תמיד, by definition, נראה תור SQS. לתור הזה תמיד יהיה dead letter Q ותמיד יהיה מוגדר הרי-try" [23:12].
- **ודווקא מנגנוני ההגנה הם שהגדילו את התקלה.** באג בפרודקשן הפעיל extraction לכל הלקוחות בבת אחת; ה-retries בכל מקום הפכו את זה לאמפליפיקציה עד שהמערכת קפאה. המסקנה: צריך גם kill switch [26:15].

2. מה זה רזיליינסי, ואיפה עובר הגבול

הסשן נפתח בשאלה מסחרית: מה חשוב יותר, למכור מיליון מוצרים בדולר או מוצר אחד במיליון דולר. אובדן של עשר מכירות בדולר לא מרגיש דרמטי, אבל אף אחד לא רוצה להיות זה שאיבד את המכירה של המיליון. המסקנה שהוצגה היא שברגע שמכניסים למשוואה מוניטין ואמון לקוחות, "כל מוצר שאנחנו מוכרים הוא חשוב. כל הודעה שהמערכת שלנו מקבלת, היא הודעה חשובה" [0:00]. זו הנחת היסוד של כל שאר ההרצאה: המערכת לא אמורה לאבד הודעות.

ההגדרה שעליה יישרו קו בפתיחה: היכולת לעמוד בתקלות תשתית, שירותים תלויים, שגיאות קונפיגורציה ובעיות רשת חולפות

מכאן עברה הדוברת ל-shared responsibility model — אותו מודל מוכר מעולם האבטחה, בגרסת רזיליינסי. AWS אחראית על ה-resilience "של" הענן: חומרה, compute, storage, database, networking, ה-Regions וה-Availability Zones. הלקוח אחראי על ה-resilience "בתוך" הענן: ארכיטקטורת ה-workload, ניהול שינויים, observability וניהול כשלים, ורשת, מכסות ואילוצים.

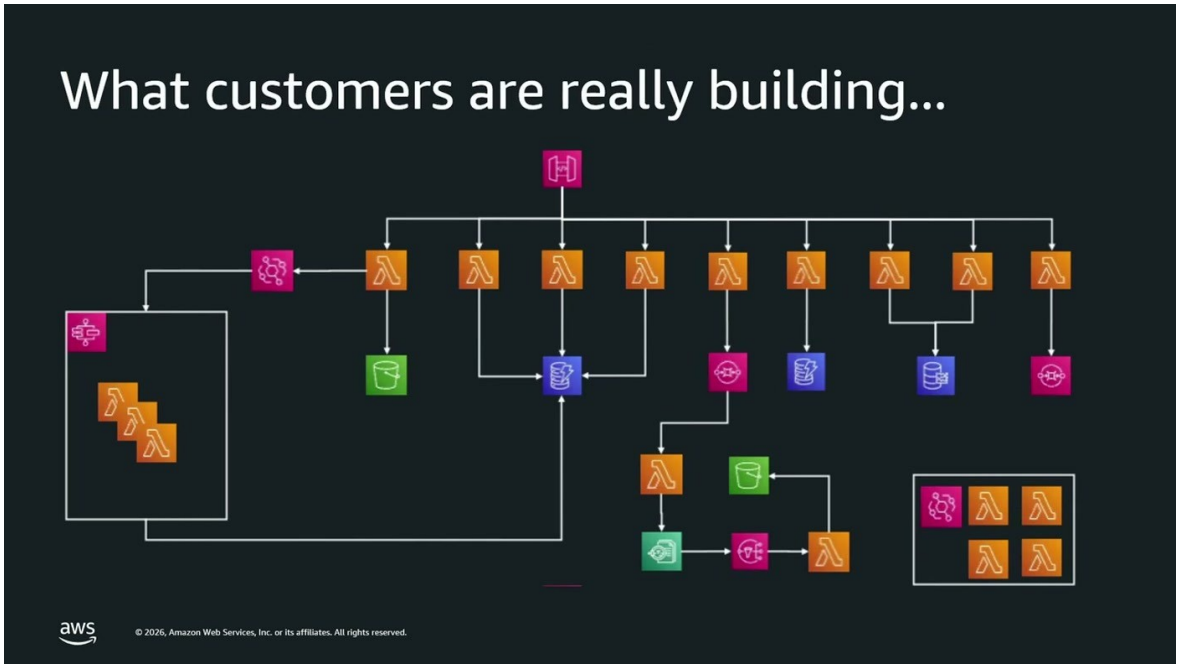


חלוקת האחריות כפי שהוצגה — שימו לב ש-workload architecture ו-quotas יושבים בצד הלקוח

כשמסתכלים על serverless, נקודת החיתוך זזה למעלה: הרבה מהאחריות התשתיתית עוברת ל-AWS, והלקוח מתפנה לעסוק בהרכבה — איך מחברים את השירותים זה לזה. אבל, וזו הנקודה שנשארה תלויה על המסך:
"רזיליאנסי אמיתי בהקשר הזה דורש בחירות ארכיטקטוניות מכוונות ברמת האפליקציה, לא רק ברמת התשתית" [3:05].

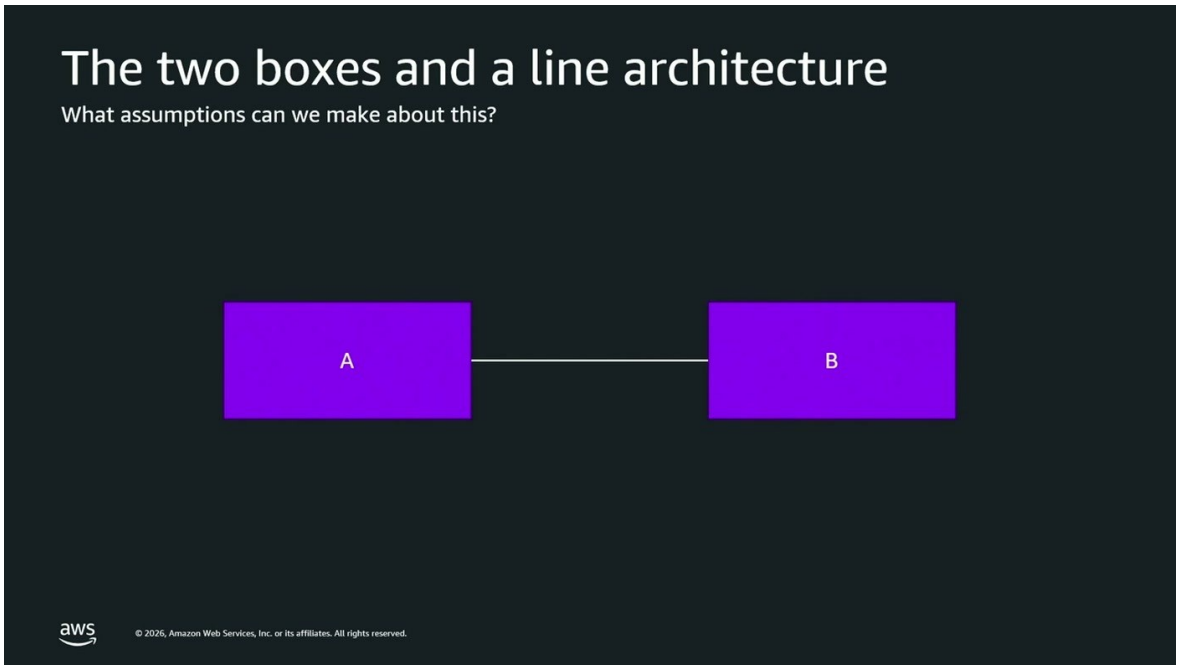
3. שתי קופסאות וקו

הדיאגרמה הקנונית של אפליקציית serverless — Client, API Gateway, Lambda, DynamoDB — הוצגה במפורש כ"מאוד מאוד נאיבית". מה שבאמת רץ אצל לקוחות נראה אחרת.



מה שלקוחות בונים בפועל: עשרות פונקציות, תורים ומאגרים, ועשרות קווים שאיש לא תיעד את ההתנהגות שלהם

כדי לתקוף את זה, הסשן צמצם את הדיאגרמה לצורה המינימלית ביותר: קופסה A, קופסה B, וקו ביניהן. התרגיל הוא לנסות לתאר מה בדיוק אנחנו יודעים על הקו הזה.



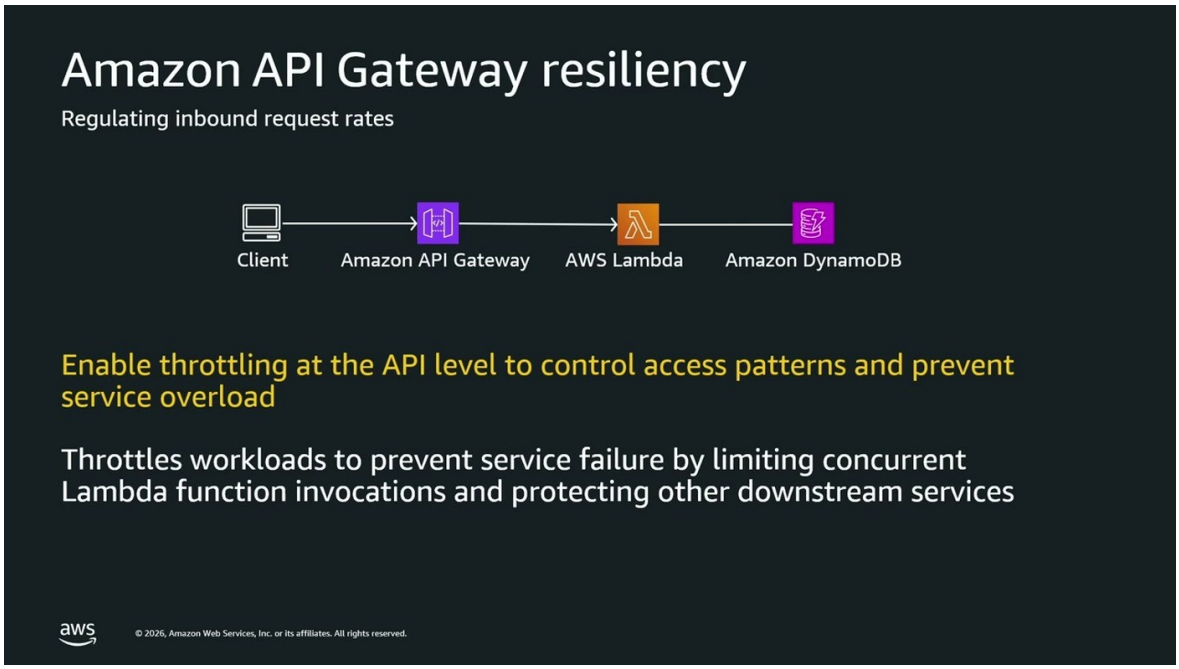
התרגיל: מה אפשר בכלל להסיק מהצורה הזאת, ומה אנחנו משלימים בראש

רשימת השאלות שהועלתה: "האם A עושה פולינג מ-B או B עושה פולינג מ-A?", האם התקשורת סינכרונית [4:35] או אסינכרונית, האם שני הרכיבים באותו חשבון, האם הם באותו Availability Zone, מה פורמט הנתונים, ומה קורה בטיפול בשגיאות — retries, exponential back-off. שום אחד מהפרטים האלה לא נמצא בדיאגרמה, וכולם קובעים אם המערכת שורדת.

מה יכול להשתבש ארבע משפחות הכשל שנמנו בפירוש לארכיטקטורות serverless: חריגה מ-limits ומכסות; קונפיגורציה שגויה; הרשאות שהוגדרו לא נכון; וזמינות משאבים וניהול state. כל אחת מהן היא החלטה ארכיטקטונית, לא תקלת תשתית.

4. Amazon API Gateway — לעצור בדלת

החלק השני של הסשן, בהגשת ה-Solutions Architect, התחיל בשער הכניסה. הרעיון המנחה: אם ידוע ש-downstream service או בסיס נתונים לא יעמדו בקצב, אין סיבה שהבקשות יגיעו עד אליהם. עוצרים אותן בשער.

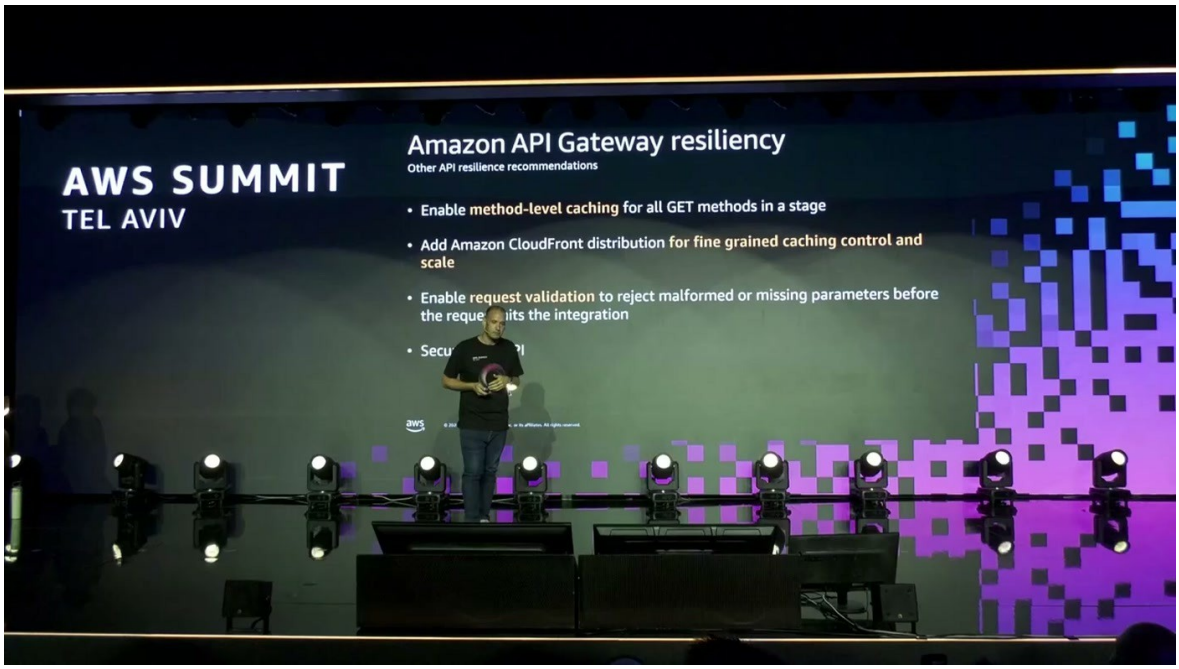


Throttling כמנגנון הגנה על השרשרת כולה, לא רק על ה-API עצמו

שלוש רמות של throttling

- **רמת החשבון —** הגנה רוחבית על כל ה-APIs בחשבון, כולל כאלה שייווצרו בעתיד, ב-steady-state או ב-burst limit.
- **רמת המתודה —** הרמה הגרנולרית שאליה כדאי לרדת: GET או POST ספציפי שמוביל לנתיב רגיש בבסיס הנתונים, עם תקרה קשיחה יותר.
- **רמת ה-API key —** שיוך usage plan למפתח לקוח. בעולמות SaaS זה מה שמאפשר להבטיח quality of service ללקוחות gold כשלקוחות free tier "יכולים להוות איזשהו noisy neighbor" [7:39].

ושלוש המלצות נוספות

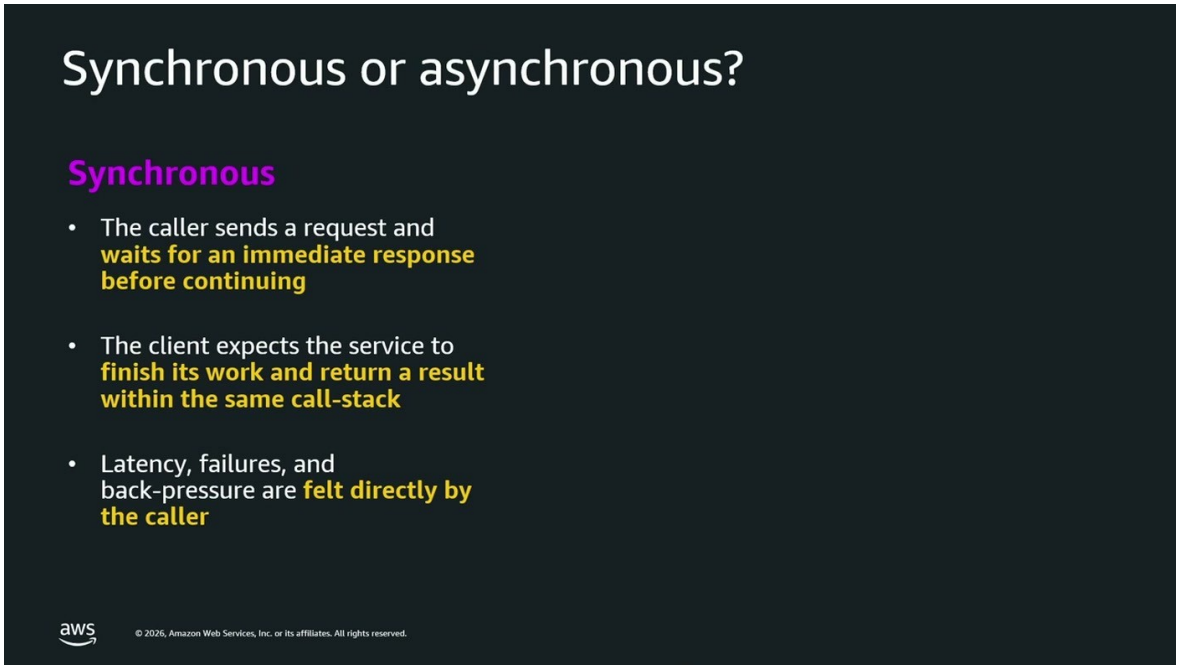


הסלייד המשלים: caching ברמת המתודה, CloudFront, request validation ואבטחת ה-API

הראשונה היא method-level caching: כשמגיעות בקשות עם אותם פרמטרים, אין סיבה לגשת שוב לבסיס הנתונים ולייצר overhead על שירותי ה-downstream — התשובה חוזרת כבר ברמת המתודה. מעליה, Amazon CloudFront נותן שליטה גרנולרית יותר: TTL לפי prefix, scale גדול יותר, והדאטה יושב ב-PoP קרוב ללקוח ולא רק ברמה הרג'יונלית. השנייה היא request validation — בקשות malformed או חסרות פרמטרים נזרקות לפני שהן נכנסות לאינטגרציה. והשלישית היא אבטחה: בקשות unauthenticated או unauthorized נעצרות בדלת.

5. סינכרוני מול אסינכרוני — יישור קו

לפני הצלילה ל-Lambda נעשתה עצירה מכוונת להגדרות, "כי הרבה פעמים יש בזה איזשהו בלבול קל" [9:11].

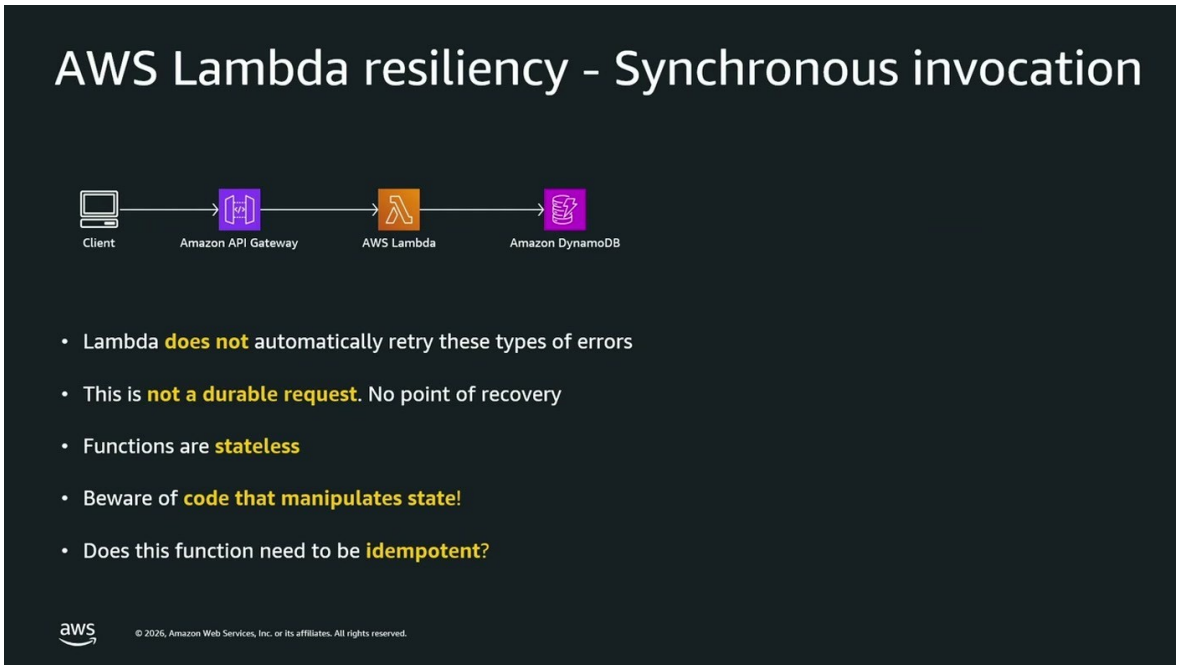


הצד הסינכרוני: הקורא ממתין לתשובה באותו call-stack, ו-latency וכשלים מורגשים אצלו ישירות

	סינכרוני	אסינכרוני
התנהגות הקורא	עוצר ומחכה שההודעה תסתיים להיות מעובדת	מקבל acknowledge (202) וממשיך
יתרון	call-stack אחיד, debug פשוט, רואים את כל הקריאות והפרמטרים	temporal decoupling בין הרכיבים
המחיר	latency מצטבר, ו-back-pressure שנופל על הקליינט	eventual consistency, נתיבי החזרה נפרדים לתוצאות ולחריגות
מה נדרש בנוסף	טיפול מפורש בשגיאות בקוד	correlation ID בכל בקשה

המודל הראשון: Synchronous Invocation (Push)

API Gateway ממשק ישירות מול Lambda, או שקוד שלכם מפעיל את הפונקציה ישירות. כאן אין רשת ביטחון.



מה שהמודל הסינכרוני לא נותן: retry אוטומטי, durability, ונקודת התאוששות

הקוד חייב לטפל בכל השגיאות בצורה graceful, כי הקליינט מחכה לתשובה. וכשהפונקציה נוגעת ב-state, נכנסת הדרישה ל-idempotency: "לא משנה כמה פעמים אני אפיל את הפונקציה שלי עם אותה הודעה ה-end state או הרזולט יהיה אותו דבר" [12:20].

הציטוט שהוצג על המסך, מתוך Enterprise Integration Patterns (Hohpe & Woolf, 2004)

6. שני מודלי ההפעלה הנותרים

Asynchronous Invocation (Event model)

כאן מקור האירוע הוא S3 Event Notifications, SNS, CloudWatch Events או EventBridge. הנקודה שהוסברה בפירוט: אתם לא מפעילים את Lambda ישירות — ההודעה נכנסת לתור פנימי שמנוהל על ידי שירות Lambda ושאתם לא רואים ולא מנהלים, ותהליך polling פנימי קורא ממנו. הקליינט מקבל acknowledge, ה-event queue מחזיר 202, ושום payload לא חוזר ללקוח.

בתמורה מקבלים טיפול מובנה בשגיאות: "יש לנו טיפול אוטומטי בשגיאות עד שתי פעמים נוספות" [13:51], עם exponential back-off כדי לא להעמיס על שירותי ה-downstream. ואם גם הניסיון השלישי נכשל — dead letter queue או destination, שמאפשרים ל-on-call לטפל בהודעה ולשמור על consistency.

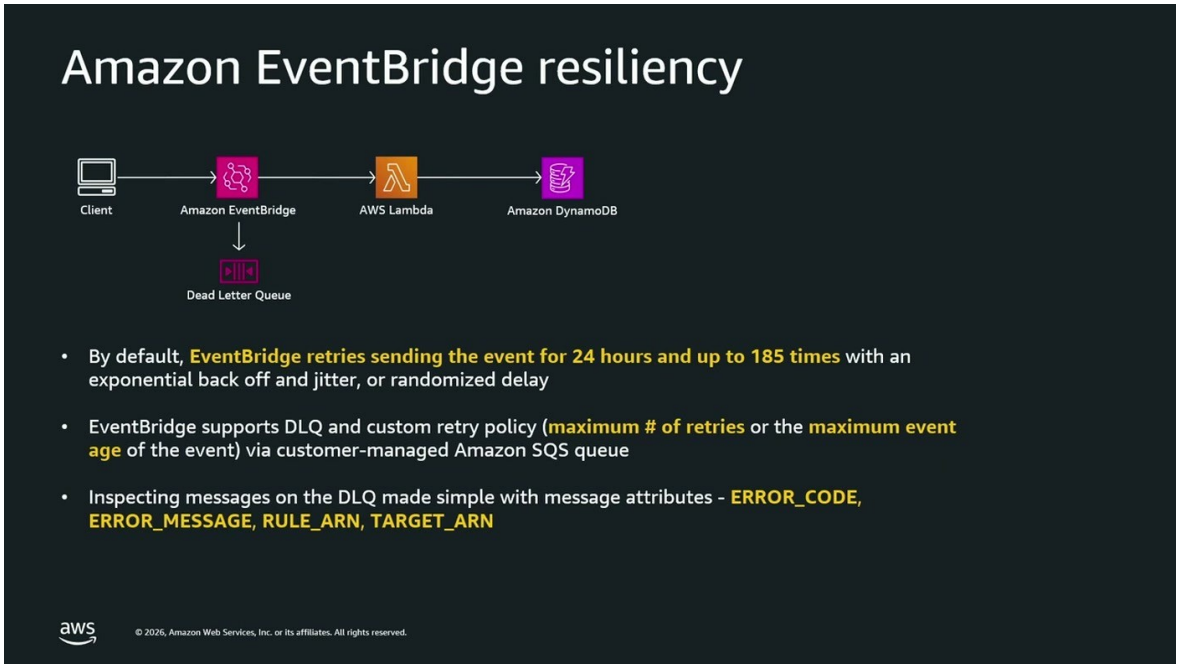
Event Source Mapping (Poll-based)

המודל השלישי עובד מול SQS, DynamoDB Streams ו-Kinesis Streams, ומכוון ל-use cases של near real-time processing בצריכת הודעות ב-batch. ההבדל המהותי: התור הוא שלכם. אתם שולטים ב-retry attempts, ב-visibility timeout ובשאר ה-levers של SQS, ו-Lambda מנהלת את ה-polling ואת ה-scale.

Partial batch failures בדוגמה שהוצגה נשלף batch של עשר הודעות והודעה מספר 6 נכשלה. מכיוון שכל ה-batch הוא invocation אחד, בלי הגדרה מתאימה כל העשר נצרכות מחדש. הפתרון: קונפיגורציה שמחזירה אובייקט response עם ה-IDs של ההודעות שנכשלו בלבד, כך שב-polling הבא מטופלות רק הן [15:21].

כדי לא לכתוב את הקוד החוזר הזה שוב ושוב, הופנה הקהל ל-Powertools for AWS Lambda — ספרייה בקוד פתוח שמטפלת ב-batching, error handling ו-retries.

Amazon EventBridge

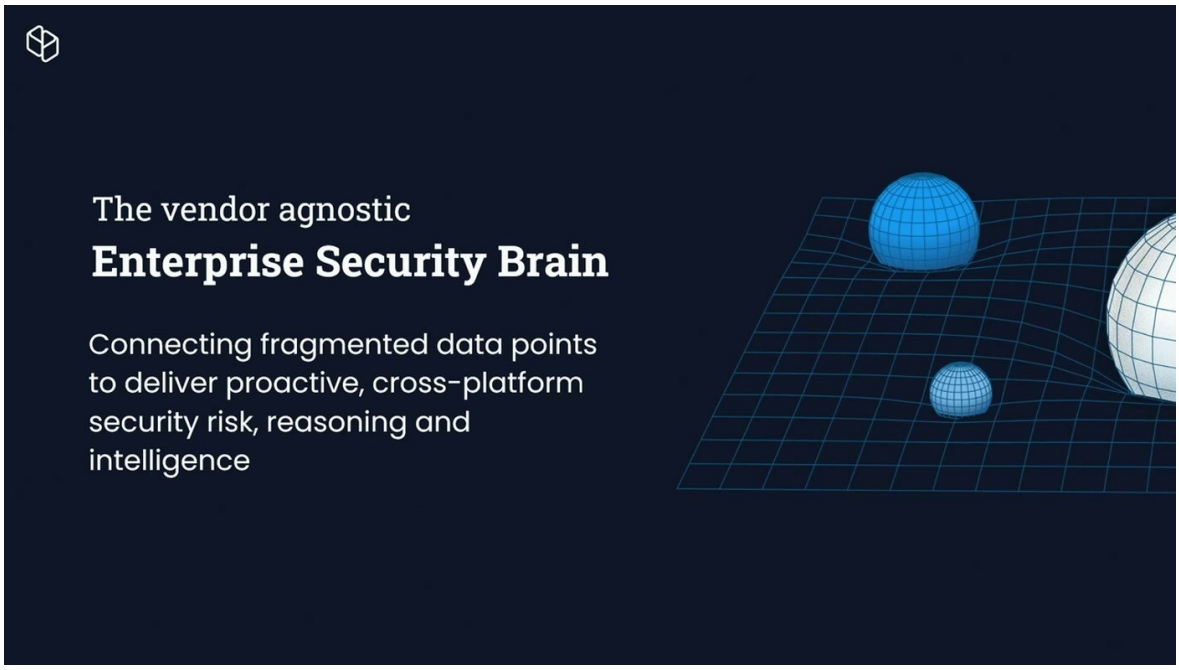


מספרי ברירת המחדל של EventBridge והמידע שמגיע עם הודעה ל-DLQ

לפי הסלייד, EventBridge שומר את האירוע ומנסה לשלוח אותו עד 24 שעות ועד 185 פעמים, עם exponential back-off ו-jitter — "אנחנו לא רוצים לייצר DDoS לעצמנו ב-downstream services" [16:57]. אפשר להגדיר retry policy משלכם (מספר ניסיונות מקסימלי או גיל אירוע מקסימלי) ולחבר DLQ מבוסס SQS. מה שהודגש: להודעות ב-DLQ מתלווים message attributes — ERROR_CODE, ERROR_MESSAGE, RULE_ARN ו-TARGET_ARN — כדי שמי שקם באמצע הלילה יידע במה מדובר.

7. Sola Security — איך זה נראה בפרודקשן

החלק השלישי הועבר ל-Principal Engineer מ-Sola Security. המוצר: שכבת קונטקסט לעולם האבטחה. "אנחנו בונים שכבת קונטקסט שמביאה את הדאטה סורסס שאנשי סיקיורטי צריכים בשביל העבודה שלהם" [18:29], ומעליה pre-computed intelligence שהופכת סוכנים — ובני אדם — למדויקים ומהירים יותר, וחסכוניים יותר בטוקנים.



מיצוב המוצר כפי שהוצג: חיבור נקודות דאטה מפוצלות לכדי הסקה וניהול סיכון חוצי-פלטפורמות

למה ETL ולא גישה ad-hoc

הדרך הפשוטה להנגיש דאטה לסוכן היא ad-hoc — הסוכן פונה ל-API החיצוני כשהוא צריך, זה מה שקורה בפועל כשמחברים MCP. הבעיה: "אנחנו מגיעים ל-rate limiting מול ה-APIs החיצוניים, וזה פשוט לא עומד בסקל" [20:06]. לכן Sola מממשת ETL: מושכים את המידע מה-API החיצוני, שומרים אותו ב-data store מקומי, והסוכנים עובדים מולו בגישה בלתי-מופרעת. בונוס: המידע מנורמל ומסודר בטבלאות, ומעליו אפשר לבנות גרף שמייצג את סביבת הלקוח, הישויות והקשרים ביניהן.

שלושת האתגרים

- **Variety —** מגוון עצום של APIs. ל-GitHub יש אותנטיקציה אחת, ל-AWS יש pagination אחר, לכל אחד error handling משלו.
- **Volume —** לא רק נפח גדול, אלא דינמי: לקוח אחד עם mono-repo אחד, ולקוח אחר עם אלפי repositories — והתשתית צריכה לספוג את ההפרש.
- **Shape —** סוגי דאטה שונים מהותית: events ולוגים מצד אחד, metadata וקונפיגורציה מצד שני.

המספרים שהוצגו

מדד	היקף
אינטגרציות יומיות על פני כל ה-vendors	מעל 20,000
רשומות ב-ingestion	מעל מיליארד
Endpoints נתמכים מ-APIs שונים	מעל 4,000
קשרים שהגרף מחשב ביום	מעל חצי מיליארד

מהארכיטקטורה הראשונה לנוכחית

ההתחלה הייתה פשוטה: Extractor שכל תפקידו לפנות ל-APIs חיצוניים, לבצע אותנטיקציה ו-pagination ולחלץ את הדאטה מגוף התשובה. הדאטה נשלח ב-batches, אבל לא דרך SQS — הקבצים נכתבים ישירות ל-S3, "פשוט כי הודעות SQS הן מוגבלות במקום" [23:12]. מה שעובר ב-SQS היא הודעה עם reference לקובץ, וה-Writer, שהוא Lambda function, קורא את הקובץ וכותב ל-data store.

הארכיטקטורה הנוכחית מורכבת יותר. EventBridge Scheduler מתזמן extraction לכל לקוח בנפרד — פעם ביום, פעם בשעה או פעם בחמש דקות לפי ה-use case. Lambda durable functions מנהלות את האורקסטרציה של workflow מורכב, עם שימוש נרחב ב-fan-out, וה-state נשמר ב-DynamoDB כי הלקוחות והסוכנים רוצים לדעת מה קורה עם ה-extraction שלהם.

> **כלל הברזל** "לפני כל רכיב קומפיוט תמיד, by definition, נראה תור SQS. לתור הזה תמיד יהיה dead letter Q ותמיד יהיה מוגדר הרי-try" [23:12]. זה מה שנותן גם רזיליינסי וגם טיפול ב-back-pressure — הדאטה זורם בצינור במקום להישפך.

בתוך ה-Writer

- **RDS Proxy מול Aurora Serverless v2 —** לא פונים לבסיס הנתונים ישירות. ה-proxy מספק connection pooling, transaction queuing ותמיכת multi-AZ.
- **SQS במצב FIFO —** ב-use cases מסוימים חשוב שהדאטה ייכתב בסדר מסוים כדי שיהיה נכון.
- **גודל הקובץ ב-S3 הוא פרמטר תכנוני —** הקובץ נטען לזיכרון של Lambda: גדול מדי מביא ל-out of memory, קטן מדי מייצר overhead ועלות מיותרת.
- **SNS לברודקאסט —** מודיע לכל הצרכנים, ובכללם הרכיב שמחשב את הקשרים בגרף, שנכתבה פיסת מידע חדשה.

הלקח: התקלה שהמנגנונים הגדילו

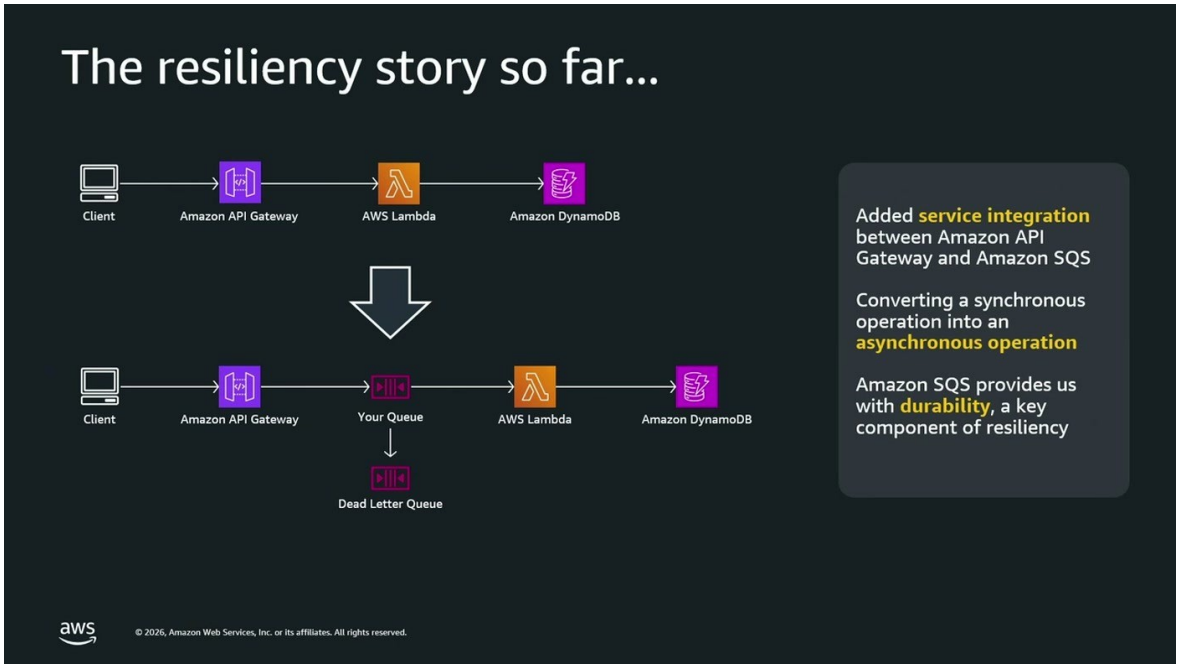
הדובר סיפר על אירוע אמיתי. בניסיון לעזור ללקוח בודד הוא ניסה להפעיל extraction לאינטגרציה אחת, "ובטעות בגלל הבג שהיה לנו פרודקשן, עשינו טריגר לכל האינטגרציות של כל הלקוחות במכה" [26:15]. בדרך כלל ה-extractions מפוזרים לאורך היום בדיוק כדי לתת למערכת אוויר לנשימה; כאן הכול יצא יחד, וה-fan-out של ה-durable functions הצטבר למיליוני הודעות בתורים.

והנה הפואנטה: "בגלל שאנחנו ילדים טובים והגדרנו ריטרייז בכל מקום ואת כל המנגנונים שעוזרים לנו לשמור על הביטחון של המערכת נעשתה אמפליפיקציה של הבעיה עד למצב שהמערכת פשוט כפאה" [26:15]. לא ניתן היה לטפל בכלום, והצוות נאלץ לנקות את התורים ידנית.

> **המסקנה המעשית** התיקון היה kill switch — gate שנמצא בכל רכיבי הפייפליין ונשען על flag (ב-DynamoDB, או בכל מנגנון דומה), שאפשר להוריד ידנית או אוטומטית. "תכינו את המערכת לגרוע מכל כי הגרוע מכל יגיע ואנחנו צריכים את הכלים האופרטיביים כדי להתמודד" [27:48].

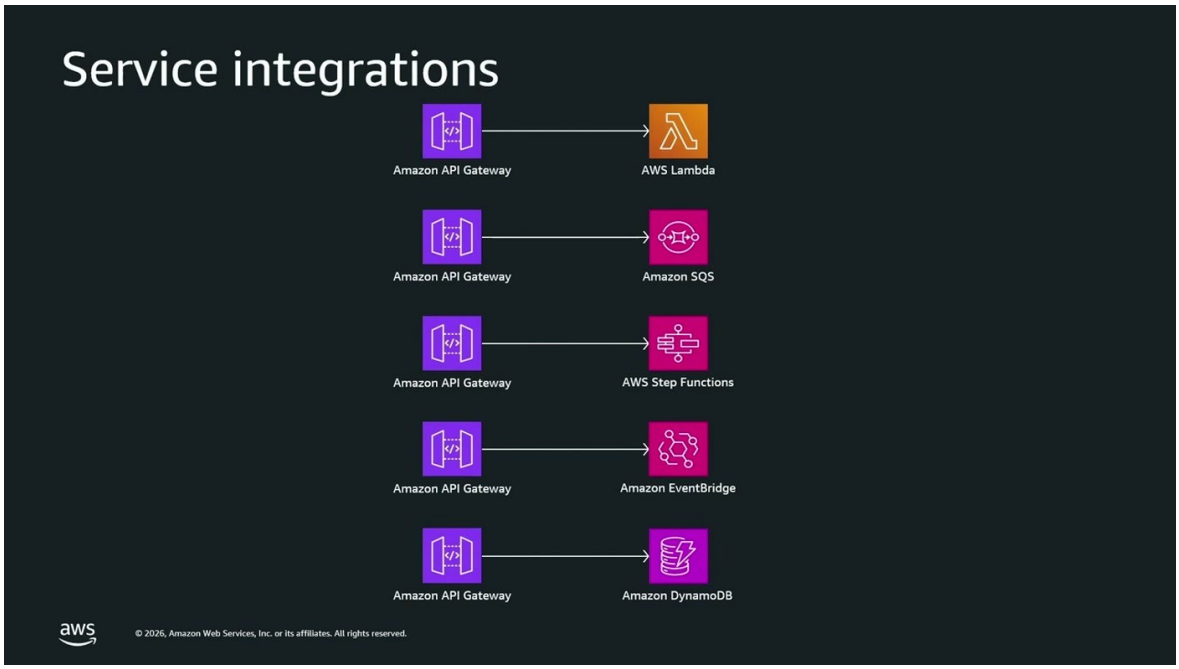
8. תבניות נוספות

החלק האחרון חזר ל-AWS והציג ארבעה כלים נוספים, שכולם עונים על אותה שאלה: מה עושים כשפעולה אחת צריכה להישאר עקבית על פני כמה רכיבים.



המעבר שנעשה עד כאן: הכנסת תור בין השער ל-Lambda הופכת פעולה סינכרונית לאסינכרונית ומוסיפה durability

Direct integrations



היעדים ש-API Gateway יכול לפנות אליהם ישירות, בלי פונקציה באמצע

API Gateway יכול לעבוד ישירות מול SQS, Step Functions, EventBridge ו-DynamoDB. מה שחוסכים בכך: "אנחנו בעצם, בכל מיני מקרים יכולים להוריד פונקציית Lambda שהיא גם עוד Compute, גם עוד כסף, גם עוד latency וגם עוד מקום שיכול לקרוס" [29:23].

> **כוכבית** לקריאת נתונים מ-DynamoDB — מומלץ ובטוח. לפעולות שמשנות state באינטגרציה ישירה צריך לחשוב פעמיים, כי "באינטגרציה ישירה כשמגיעים ישר מ-API Gateway זה הופך להיות בעיה של הקליינט" [29:23].

פירוק לפעולות, ו-DynamoDB Streams

הפירוק שהוצג: שלב ראשון — הכנסת ההודעה לתור; אם נכשל, הקליינט מקבל שגיאה והמערכת נשארת consistent כי שום דבר לא השתנה. שלב שני — קריאה מהתור וכתיבה ל-DynamoDB, עם מנגנון ה-retries של SQS ו-DLQ מאחוריו. אבל השלב השלישי הוא מקור הבעיה הקלאסי: "מה קורה אם שלחתי הודעה שבוצע מחירה אבל לא הצלחתי לכתוב לדאטאבייס או לחלופין כתבתי לדאטאבייס אבל לא הצלחתי לשלוח את ההודעה" [30:57].

ההצעה: במקום שפונקציה אחת תעשה שני דברים, משרשרים פעולה. מפעילים DynamoDB Streams — רצף האירועים שקורים בבסיס הנתונים — וקוראים ממנו באמצעות EventBridge Pipes, שמעביר הלאה ל-EventBridge. שני היתרונות של Pipes שהודגשו: filtering, כי לא כל השינויים מעניינים; ו-enrichment, כי "ההודעה שמגיעה זה הודעה של DynamoDB, אין לה קונטקסט עסקי" [32:28] — וצריך לתרגם אותה לאירוע שמשמעותי למשתמשי הקצה.

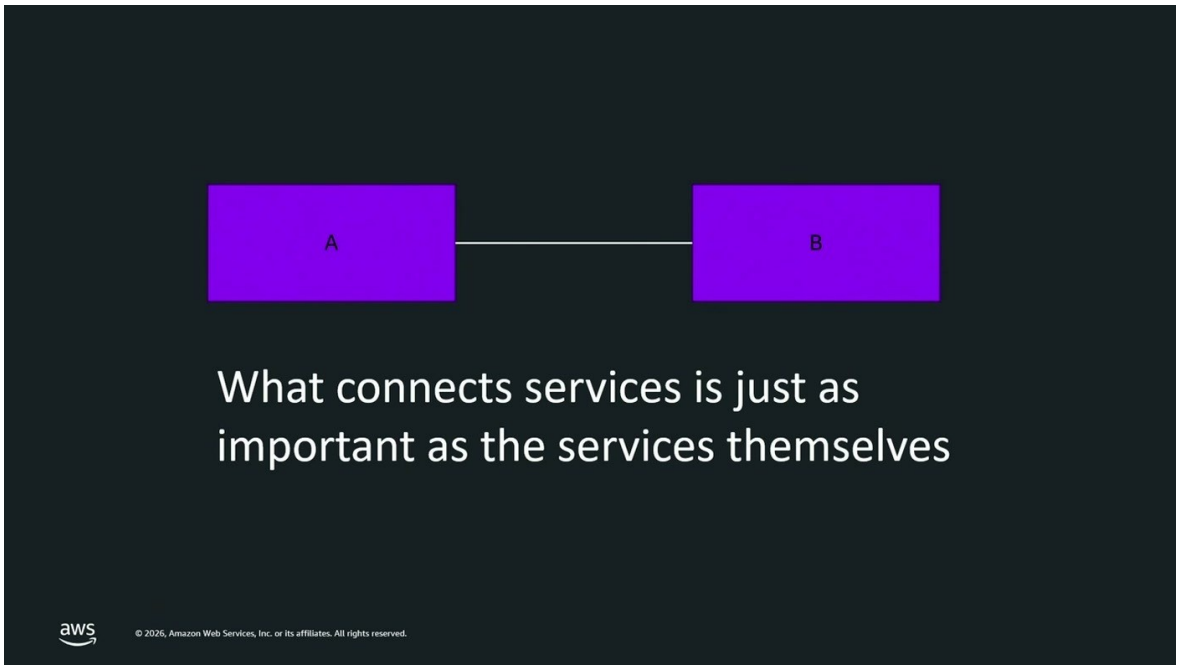
Step Functions

האפשרות השנייה לאורקסטרציה. היתרונות שנמנו: ניהול שגיאות מובנה, retries ו-exponential back-off כחלק מהשירות עצמו ולא בקוד, ו-timeouts ו-heartbeats לתהליכים ארוכים — כך שאפשר להכשיל בצורה מבוקרת, ואחר כך לראות מה נפל, איפה נפל, ולהחזיר את ההודעה שנפלה במקום לאבד אותה.

Saga pattern

התבנית האחרונה, לשמירת consistency בתהליך שמפוצל לתתי-משימות: לכל משימה מוגדרת משימה נגדית שמחזירה את המערכת למצב עקבי. הדוגמה: הזמנת חופשה — מלון, טיסה, רכב. אם אחד השלבים נכשל, לא רוצים לסיים את התהליך "בהצלחה" חלקית, ולכן מוחקים את פרטי הטיסה ומבטלים את הזמנת המלון שכבר הצליחה. "יש לנו משימה נגדית שיכולה לקחת את המערכת ולהחזיר אותה למצב הקונסיסטנטי" [34:00] — נכשלים בצורה בטוחה.

9. מה לוקחים מכאן



המשפט שסגר את המעגל לתרגיל שתי הקופסאות והקו

- **מה שמחבר בין השירותים חשוב כמו השירותים עצמם.** הקו בדיאגרמה הוא לא פרט טכני — הוא מקום הכשל השכיח.
- **בחרו את מודל ההפעלה לפי התנהגות הכשל, לא לפי הנוחות.** סינכרוני בלי retry אוטומטי ובלי נקודת התאוששות הוא החלטה, לא ברירת מחדל.
- **idempotency היא דרישה, לא שיפור.** בכל מסלול שיש בו retry — ואסינכרוני ו-poll-based הם כאלה בהגדרה — הפונקציה תופעל פעמיים על אותה הודעה.
- **DLQ בלי מי שקורא אותו הוא לא רזיליינסי.** ההודעות מגיעות לשם עם ERROR_CODE ו-ERROR_MESSAGE בדיוק כדי שאפשר יהיה לטפל בהן; זה תהליך אופרטיבי, לא הגדרה.
- **retries בלי kill switch מגדילים תקלה.** זה הלקח היקר ביותר בסשן: אותם מנגנוני הגנה שמצילים ממקרים קטנים מאמפלפים מקרים גדולים.
- **אתגרו את ההנחות בפעם הבאה.** "פעם הבאה שאתם בונים ארכיטקטורה תסתכלו על האפליקציה שלכם תסתכלו על ההנחות שבאות לכם כמובן מאליו ותנסו לאתגר אותם" [35:33].

מקורות שהופנו אליהם

- Serverless Land (serverlessland.com) — תבניות, דוגמאות קוד וחומרי לימוד ל-serverless ב-AWS
- Powertools for AWS Lambda — ספריית קוד פתוח ל-batching, error handling ו-retries, זמינה דרך אותו אתר
- Serverless Track — סדרה בת שישה מפגשים במשרדי AWS, ללא עלות

10. מונחון

מונח	משמעות
Shared responsibility model	חלוקת האחריות בין AWS (ה-resilience "של" הענן) ללקוח (ה-resilience "בתוך" הענן)
Throttling	הגבלת קצב בקשות בשער כדי להגן על שירותי downstream; ברמת חשבון, מתודה או API key
Usage plan	שיוך מכסות ל-API key, לרוב לפי tier של לקוח
Idempotency	תכונה שבה הפעלה חוזרת של אותה פעולה עם אותה הודעה מותירה את אותו end state

מונח	משמעות
Event Source Mapping	מודל poll-based שבו Lambda שואבת הודעות ב-batch מתור או stream שבבעלותכם
Partial batch failure	החזרת ה-IDs של ההודעות שנכשלו בלבד, כדי שלא לעבד מחדש את כל ה-batch
DLQ	Dead Letter Queue — יעד להודעות שלא הצליחו להיות מעובדות אחרי כל הניסיונות
Exponential back-off + jitter	הגדלה מעריכית של המרווח בין ניסיונות, עם השהיה אקראית שמונעת גלי retry מסונכרנים
EventBridge Pipes	שירות שקורא ממקור אירועים, מסנן, מעשיר ומעביר ליעד — כאן מ-DynamoDB Streams ל-EventBridge
Saga pattern	תבנית שבה לכל שלב בתהליך מוגדרת פעולת פיצוי שמחזירה את המערכת למצב עקבי
Back-pressure	העומס שחוזר אחורה במערכת כשהצרכן איטי מהיצרן
RDS Proxy	שכבת proxy מנוהלת לבסיס נתונים: connection pooling, transaction queuing ותמיכת multi-AZ