Agentic AI ו-Generative על Amazon EKS

AWS Summit Tel Aviv 2026 ,סיכום סשן — TLV-SVS303

דוב אמיר (Senior Solutions Architect, AWS) · ארז צרום (AWS) · יובל רבובסקי (Kaltura)

10 בספטמבר 2026 | משך: כ-43 דקות | הופק מהקלטת הסשן

מסלול: AWS for Containers and Platform Engineering | רמה: 300

על המסמך הזה

המסמך מסכם את הסשן TLV-SVS303 מתוך AWS Summit Tel Aviv 2026, בהתבסס על ההקלטה המלאה. הוא נבנה מתמלול אוטומטי של ההקלטה ומהסליידים שחולצו מהווידאו עצמו, כך שאפשר לחזור לתוכן בלי לצפות שוב בארבעים ושלוש הדקות. חותמות הזמן בגוף המסמך מפנות לנקודה המדויקת בהקלטה.

הסשן מחולק לשלושה חלקים ברורים: סקירת ההכרזות של EKS לעומסי AI (דוב אמיר), צלילה טכנית לאג'נטים ול-inference (ארז צרום), וסיפור פרודקשן אמיתי מ-Kaltura (יובל רבובסקי). המסמך שומר על אותו סדר.

איך לקרוא את זה

- **פרק 1 — תקציר מנהלים:** חמש דקות קריאה. מה הוצג, ומה שווה לבדוק אצלכם.
- **פרקים 2–5 — מה חדש ב-EKS:** קומפיוט, DRA, Karpenter, control plane ו-MCP Server.
- **פרקים 6–8 — אג'נטים:** Strands Agents, defense in depth, ופרויקט Agent Sandbox.
- **פרקים 9–11 — Inference:** ארכיטקטורה טיפוסית, האצת טעינת מודלים, ו-SOCI.
- **פרקים 12–14 — Kaltura בפרודקשן:** האבטארים בזמן אמת, cold start, ושיתוף GPU.
- **פרק 15 — מה לוקחים מכאן:** מסקנות ונקודות פעולה, ואחריו מילון מונחים.

הערת מקור המסמך הופק מתמלול אוטומטי של ההקלטה. התמלול מדויק בקטעים המקצועיים, אך מונחים לועזיים מופיעים בו בשיבוש פונטי (למשל "סוצ'י" = SOCI, "סטרנץ" = Strands, "קרפנטר" = Karpenter). בגוף המסמך המונחים מנורמלים לאנגלית; בציטוטים הם נשארו כפי שנאמרו. הציטוטים אומתו מול ההקלטה בחותמת הזמן המצוינת. מספרים שמופיעים על הסליידים צוינו ככאלה.

1. תקציר מנהלים

שלושה דוברים לקחו את אותו קו טיעון משלוש זוויות: AWS טוענת ש-EKS כבר אינו רק פלטפורמת מיקרו-סרוויסים אלא תשתית ה-AI המרכזית; ארז צרום פירק את זה למנגנונים קונקרטיים לאג'נטים ול-inference; ויובל רבובסקי מ-Kaltura הראתה את אותם מנגנונים עובדים בפרודקשן, עם מספרים.

- **הכאב האמיתי בהרצת AI על Kubernetes הוא cold start, לא אימון.** אימג'ים של מנועי inference שוקלים 8–10 ג'יגהבייט לפני שהוסיפו מודל, ו-Kaltura חיכו 8–12 דקות לפוד שעונה ב-real time ברגע שהוא למעלה [30:49].
- **הפתרון מפוצל לשניים: להאיץ את האימג' ולהוציא ממנו את המודל.** SOCI מקביל את ההורדה ואת פריסת השכבות; Run:ai Model Streamer ו-NVIDIA ModelExpress מקבילים את טעינת המשקולות מ-S3 אל ה-CPU ואל ה-GPU.
- **הנתון החזק ביותר בסשן הוא ניצולת ה-GPU.** Kaltura גילו שהם עמדו על 10% ניצולת ממוצעת [37:04]; אחרי מעבר ל-MPS דרך Node Overlay של Karpenter הם על 87% בממוצע — שיפור של מעל פי 8, בלי לוותר על זמני תגובה בזמן אמת.
- **אג'נטים הם workload אחר, וה-best practice השתנה.** דוב אמיר: הרצת אג'נטים בקונטיינר רגיל כבר אינה ברירת המחדל — מיקרו-VM כמו Kata Containers הוא הסטנדרט, ו-Karpenter יודע כעת לנתב פוד כזה לאינסטנס שתומך ב-nested virtualization [7:51].
- **Agent Sandbox נותן ל-Kubernetes את מה שחסר לאג'נטים.** פרויקט OSS תחת SIG Apps, שעבר ל-beta, עם CRD שנותן זהות יציבה, singleton, suspend/resume, ו-warm pool שמגיש פוד חם תוך פחות משנייה — נדרש כשלקוחות מבקשים בסביבות אלף פודים בשנייה [17:07].
- **הפלטפורמה שנבנתה ל-inference שימשה אחר כך לכל AI בארגון.** Kaltura מריצים על אותה תשתית גם את האבטארים, גם אג'נטים פנימיים (NOC ו-FinOps) וגם אג'נטים מול לקוחות [40:07].

2. למה EKS, ומה עומד בדרך

דוב אמיר, ארכיטקט בכיר ב-AWS, פתח בהצבעה באולם — מי מריץ AI על EKS, בין אם training, inference או אג'נטים — וקיבל תגובה רחבה. הטענה שלו: EKS אינו עוד פלטפורמה למיקרו-סרוויסים אלא תשתית מרכזית ל-AI, ובכל שבוע רצים עליו מיליוני שרתי GPU.

> **— דוב אמיר, [0:00]** למה דווקא על קוברנטיס? כי רק על קוברנטיס יש את הגמישות, את הסקיילביליות, ואת הפתיחות לפתרונות האופן סורס שהכרחים לפרויקטים מורכבים של AI.

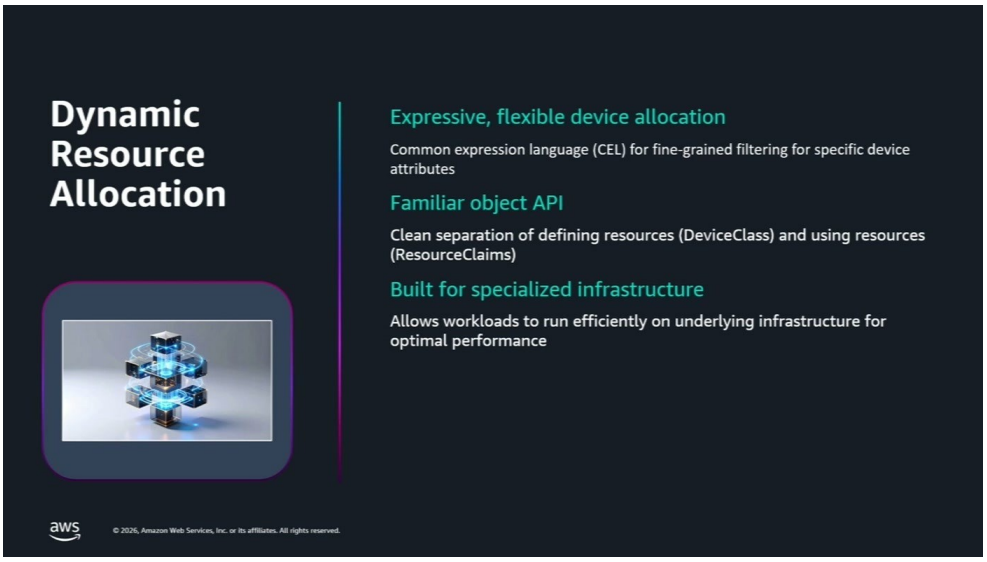
חמשת האתגרים שהוצגו מהשטח: זמינות GPU, אופטימיזציית עלות, ניצולת GPU, מורכבות הנוף הטכנולוגי ופערי ידע.

מיד אחרי זה הוא הוריד את הטון: "להריץ AI על EKS זה לא [כל] כך פשוט, יש מספר אתגרים" [1:38]. שני האתגרים שהוא בחר להרחיב עליהם הם זמינות קומפיוט — "אנחנו רוצים להריץ את ה-GPU הנכון, בריג'ן הנכון ובזמן הנכון" [1:38] — ויעילות. הדרישה לגבי יעילות נוסחה כדרישת ארכיטקטורה: כל הסטאק חייב להיות GPU aware, עם scheduling מדויק, scaling מהיר, ויכולת מדידה מרמת הפרויקט דרך האינסטנס ועד הפוד הבודד.

3. קומפיוט, AMI ו-Dynamic Resource Allocation

בצד החומרה דוב סקר את קו האינסטנסים החדשים: P6E, GB200 ו-GB300 כאינסטנסים הגדולים ביותר, מרובי GPU, המתאימים לאימון מודלי שפה גדולים; G7E כאינסטנסים קטנים יותר, מצוינים ל-inference ולגרפיקה; ולצידם המאיצים שפותחו ב-AWS — Trainium ו-Inferentia — שלדבריו פותחו בישראל בחטיבת Annapurna Labs. אינסטנס נוסף שהוזכר הוא G6F, כאשר ה-F הוא fractional: הרצת מודלים קטנים על חלק מ-GPU כדי לחסוך בעלות [3:15].

מעל החומרה יושבת בעיית התאימות. במקום שכל צוות ירכיב לעצמו דרייברים של NVIDIA, מודולי קרנל וחבילות תוכנה ויתמודד עם תאימות גרסאות, AWS מספקת את ה-EKS Optimized GPU AMIs עם כל זה מובנה ובדוק.



DRA: הקצאת מכשירים דרך CEL וסינון עדין לפי מאפייני מכשיר, עם הפרדה בין DeviceClass לבין ResourceClaim.

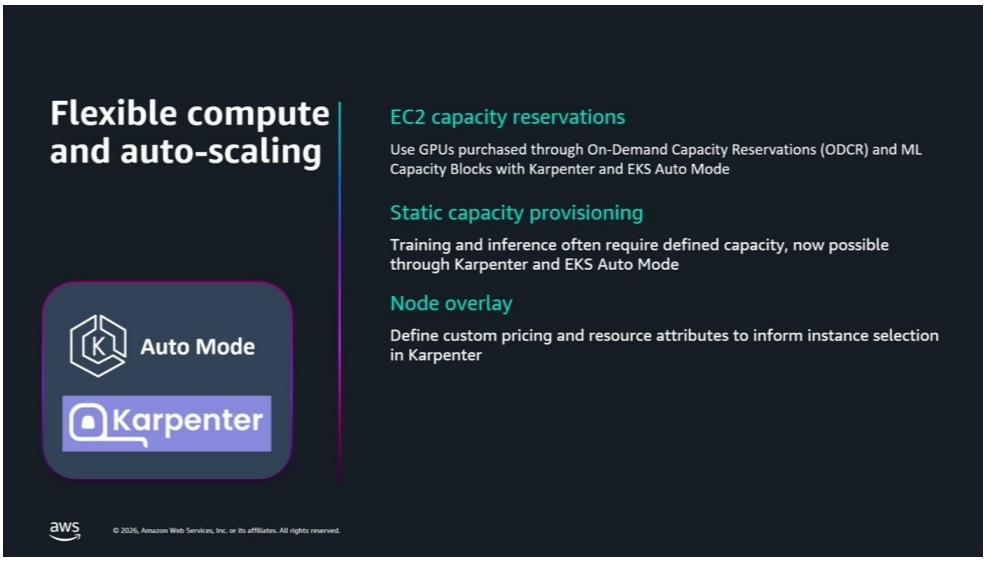
החידוש המשמעותי בשכבת ה-scheduling הוא Dynamic Resource Allocation — DRA — שנתמך החל מגרסת EKS 1.33 [3:15]. DRA מחליף בהדרגה את ה-device plugins המסורתיים. בעולם הישן ההקצאה הייתה קשיחה: פוד אחד מקבל GPU אחד. ב-DRA ההקצאה דינמית ומבוססת claims, באותו רעיון שבו מוקצה storage לפוד דרך PersistentVolumeClaim.

> **— דוב אמיר, [4:45]** די-אריי בעצם מאפשר לגדיר לפוד איזה סוג של ג'י-פיו צריך כמה ג'י-פיו ואיזה סוג של נטרורקינג הוא צריך וגם אפשר לגדיר בקלות חלוקה של ג'י-פיו ככה שפוד יעבוד על חלק של ג'י-פיו ולא ייקח GPU שלם עם מגנונים כמו מיגס ו-MPS.

החיבור הזה בין DRA לבין MIG ו-MPS הוא בדיוק הקו שיובל רבובסקי תמשוך אליו בהמשך — שם זה יהפוך למספר.

4. Karpenter: capacity, בידוד ו-placement

לפני Karpenter הוזכר SOCI כפתרון לזמן עליית הפוד (ראו פרק 11), אבל כדי ש-scaling יעבוד לא מספיק שהפודים יעלו מהר — גם המכונות צריכות. Karpenter, שפותח ב-AWS ונתרם ל-CNCF, כבר עובד לא רק על EKS אלא גם על קלאסטרים אחרים. דוב לא נכנס ל-deep dive (הייתה הרצאה נפרדת לפניו) אלא מנה חמישה פיצ'רים חדשים שרלוונטיים ל-AI.



שלושת הפיצ'רים הראשונים: EC2 capacity reservations ו-static capacity provisioning, capacity blocks, ו-Node Overlay.

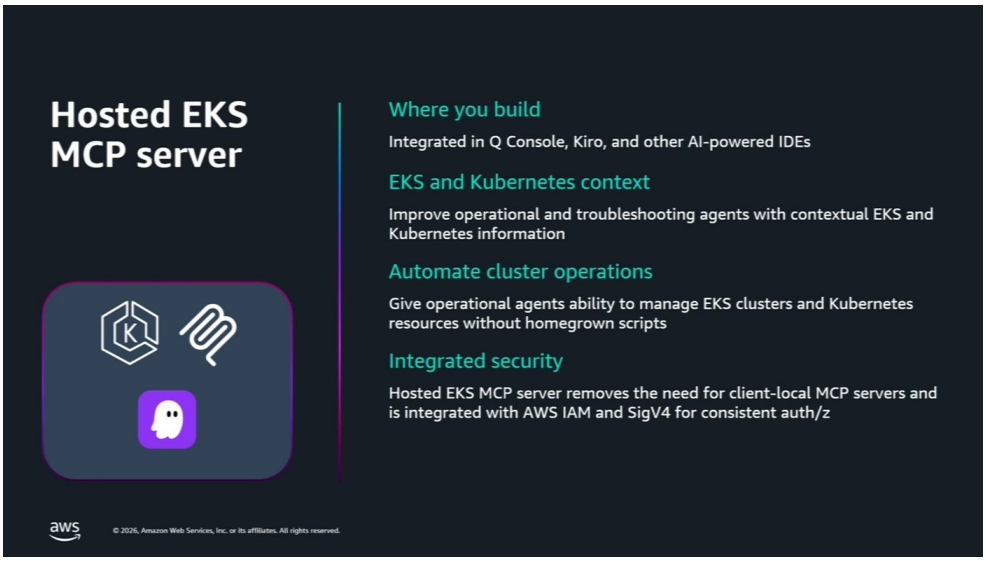
- **Capacity reservations ו-capacity blocks.** הזמנה מראש של GPU לזמן מוגדר — "אתם יכולים להגיד שביום שלישי הבא אתם צריכים מכונה מסוימת ואז מובטחת לכם הזמינות" [6:18]. Karpenter מכיר כעת את ההזמנות האלה ויודע לצרוך אותן.
- **Static capacity provisioning.** לפעמים, בעיקר ב-training, דווקא לא רוצים scale up ו-down אלא קלאסטר בגודל קבוע. עכשיו זה אפשרי.
- **Node Overlay.** השפעה על מנגנון ה-scheduling של Karpenter, כולל הגדרת חלוקת GPU כך שפוד ירוץ על חלק ממנו. דוב הפנה כאן במפורש להמשך הסשן — וזה אכן הכלי שבו Kaltura השתמשו.
- **תמיכה ב-nested virtualization.** היכולת של EC2 להריץ VM בתוך EC2 — נתמך רק באינסטנסים חדשים.
- **Placement groups.** אסטרטגיית clustering מקרבת אינסטנסים לאותו network partition לתקשורת מהירה במיוחד (שימושי ב-training); אסטרטגיית spread מפזרת אותם ככל האפשר לשיפור שרידות [7:51].

הפיצ'ר הרביעי הוא זה שמסמן שינוי תפיסתי, ולא רק יכולת:

> **— דוב אמיר, [7:51]** היום הבס פרקטיס הפך להיות שכשמריצים AI agents בתוך קוברנטיס, הבס פרקטיס הוא להשתמש במייקרו-VM, כמו למשל קאטה קונטיינרס, ולא בקונטיינר רגיל. למה עדיף מייקרו-VM? כי זה מייצר בעצם סיינדבוקסינג ובעצם בידוד הבטחתי יותר חזק לאיג'נט.

5. Control plane, MCP Server ו-AI Conformance

כשקלאסטר מגיע לאלפי אינסטנסים, גם ה-control plane צריך לגדול. Provisioned Control Plane מאפשר להגדיל אותו באופן יזום ומראש — למשל לפני ריצת training גדולה — במקום לחכות שיסתגל [9:25].



Hosted EKS MCP Server: אינטגרציה בסביבות פיתוח, הקשר EKS/Kubernetes לאג'נטים, אוטומציה של פעולות קלאסטר, ואבטחה דרך IAM ו-SigV4.

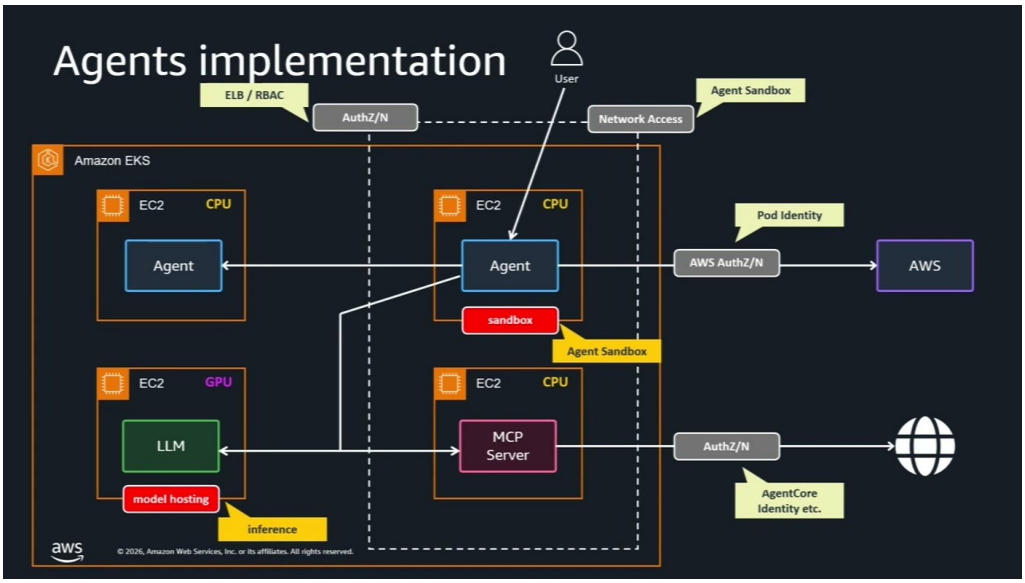
בשכבת התפעול הוצג ה-Hosted EKS MCP Server, שנמצא ב-preview: MCP Server מנוהל בצד AWS, עם שכבת ההרשאות של RBAC ו-IAM, שמאפשר לדבר עם הקלאסטר באנגלית או לפתח DevOps agents שמושכים מטריקות ולוגים ומטפלים בתקלות. הסלייד מדגיש שהוא מסיר את הצורך ב-MCP Server מקומי בצד הלקוח.

את חלקו דוב סגר בטענה שכל ההכרזות האלה יחד מסתכמות בכך ש-EKS עומד בתקן Kubernetes AI Conformance, ובמשפט שסיכם את קצב השינוי:

> **— דוב אמיר, [9:25]** לפני שנה להריץ מודל של מיליארד פרמטרים על EKS היה פרויקט של חודשים אפשר רק לדמיין איפה נהיה בשנה הבאה

6. אג'נטים כ-workload: defense in depth

ארז צרום פתח בהנחה שאג'נט הוא חיה חדשה עם דרישות שונות. ברמת ה-high level הארכיטקטורה מוכרת: agent loop שמדבר עם משתמש, פונה ל-LLM אצל provider כלשהו (למשל Bedrock), ומחובר לכלים ול-MCP servers. מה שמשנה את כללי המשחק הוא שהאג'נט אינו דטרמיניסטי ויכול לפעול אוטונומית — ומכאן האתגרים.



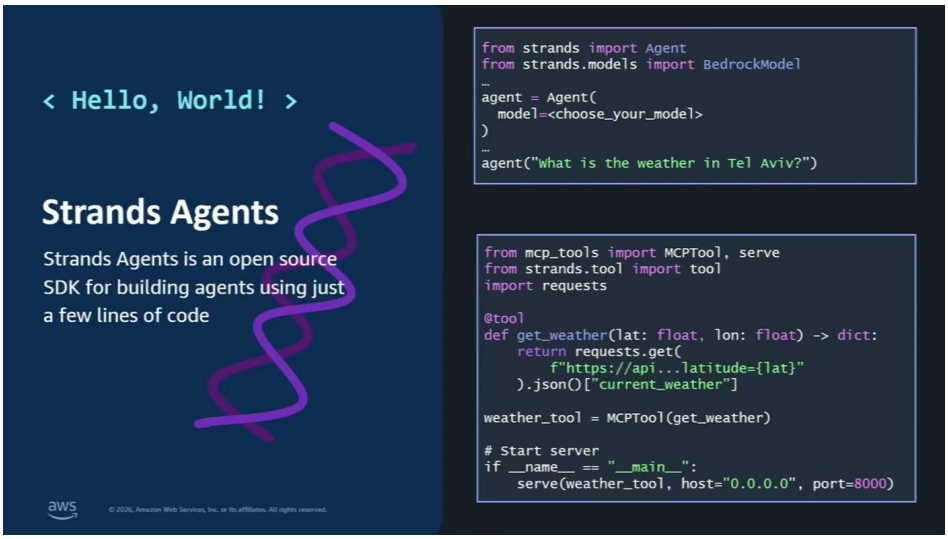
מימוש אג'נט על EKS עם שכבות הגנה: AuthN/Z בכניסה, Network Policy, Pod Identity מול שירותי AWS, וה-agent בתוך sandbox נפרד מה-MCP Server וה-LLM.

- **Least privilege דרך Pod Identity.** אם האג'נט צריך לדבר עם שירות AWS — "אני יוכל בעצם להשתמש בפוד identity כדי לתת לו את הפרמישנים הכי מינימליים" [12:25].
- **צמצום blast radius.** ההנחה היא שהאג'נט עלול לנסות לצאת מהסביבה שלו, ולכן התכנון מתחיל מהשאלה מה קורה כשזה קורה.
- **Zero trust פנימי.** אם האג'נט חוקר את סביבתו, הוא לא אמור להגיע לשירותים פנימיים שאין לו סיבה לדבר איתם. הכלי שהוצע: VPC CNI עם תמיכה ב-Network Policies.
- **Sandboxed execution.** runtime כמו Kata Containers מעל Firecracker, כהמשך ישיר ל-nested virtualization שדוב הציג.

> **— ארז צרום, [12:25]** ובעצם כל פוד הוא מיקרו-VM בפני עצמו ככה שגם אם האג'נט הנסה לצאת מהסביבה שלו למשל עם טכניקות של container escape, הוא לא יוכל לגשת לאוס את עצמו.

7. Strands Agents: איך נראה הקוד

כדי להראות כמה קרוב זה לקוד רגיל, ארז הציג את Strands Agents — SDK אופן-סורס של AWS לבניית אג'נטים. בחלק העליון של הסלייד agent loop בסיסי שנשאל מה מזג האוויר בתל אביב; בחלק התחתון, אותה ספרייה חושפת MCP Server: פונקציה שקוראת ל-API של מזג האוויר, עטופה ב-decorator של tool.

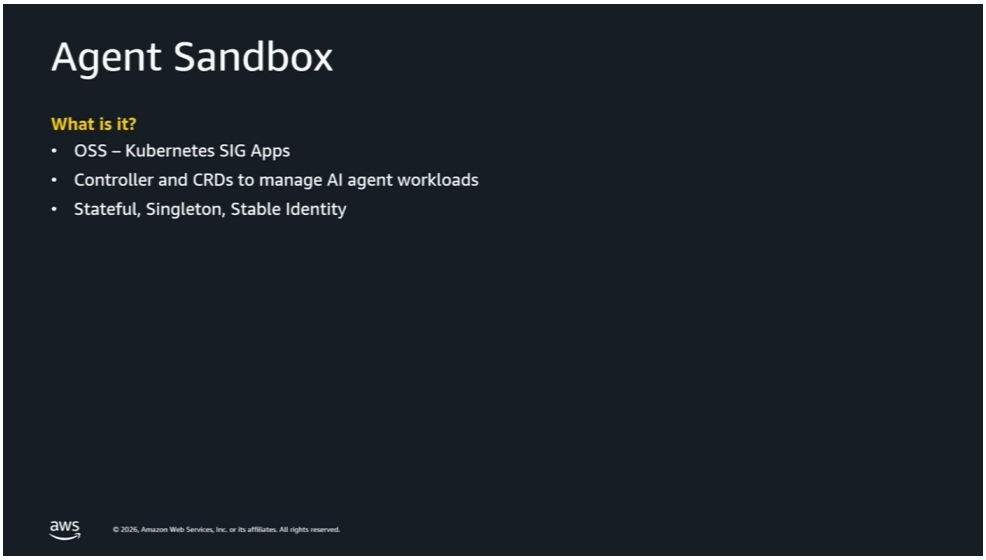


שני הקטעים על אותו סלייד: יצירת Agent מול BedrockModel, ומתחת — עטיפת פונקציה כ-tool והרצתה כ-MCP server על פורט 8000.

הנקודה המעשית: "אנחנו לא צריכים להכיר את כל הידע וכל הפרוטוקול של [13:57] "mcp. את התוצאה עוטפים בקונטיינר ומריצים על EKS — וכאן עוברים לשאלה איך מריצים את זה בסקייל.

8. Agent Sandbox: זהות יציבה ו-warm pool

Agent Sandbox הוא פרויקט אופן-סורס תחת Kubernetes SIG Apps, שעבר לאחרונה graduation ל-beta. הוא מורכב מ-controller ומכמה CRDs, ומביא שלוש תכונות: stateful, singleton, ו-stable identity.



הגדרת הפרויקט כפי שהוצגה: OSS תחת Kubernetes SIG Apps, controller ו-CRDs לניהול עומסי אג'נטים.

ארז הסביר את הפער דרך מה שקיים היום. ב-Deployment הזהות היא שם ועוד סיומת אקראית, כי המטרה היא רפליקציה של פודים. ב-StatefulSet הזהות ממוספרת — שם ועוד אינדקס. לאג'נט צריך משהו שלישי: "אני אוכל לייצר פודים שהם מוגדרים כ-stateful מהצד של ה-control plane אבל הישות שלהם יהיה שם קבוע שאני אוכל להגדיר אותו מראש כמו למשל session id מסוים" [15:33]. השם הזה מחלחל לכל מה שהפוד מחזיק — PVCs, service account וכו'.

ה-CRD המרכזי נקרא Sandbox: שם (שהוא גם הזהות היציבה), RuntimeClass (למשל Kata מעל Firecracker), אימג', PVC ו-Network Policies. מעליו יש extensions — למשל SandboxTemplate, תבנית מוכנה מראש עם resources, PVC ו-runtime class.

- **Singleton.** כל אג'נט הוא single instance; גם אם יש כמה sub-agents, כל אחד מהם אחיד בפני עצמו.

- **Suspend ו-resume מהירים.** רוב האג'נטים idle רוב הזמן. היכולת להשעות ולחדש במהירות מאפשרת multiplexing על אותו node וחיסכון בעלות ובמשאבים [15:33].
- **זיהוי idle אוטומטי.** בפיצ'ר שנוסף לאחרונה, ה-controller מנטר את ה-cgroup ומבין אם יש תזוזה במשאבים; אם אין — הוא משעה את הפוד לבד [18:39].

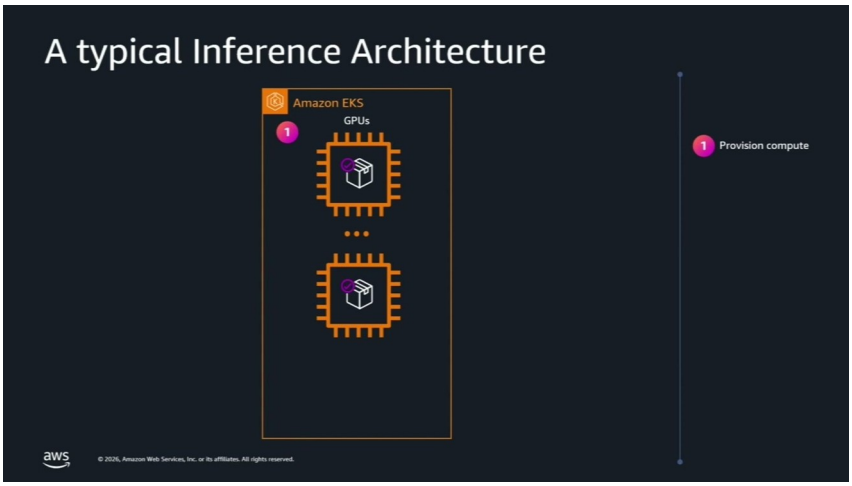
הבעיה שה-warm pool פותר היא בעיית קצב, והיא הוצגה עם מספר:

> **— ארז צרום, [17:07]** למשל יש לקוחות שמרימים סביבות להרצת קוד, אז אנחנו רואים באזור האלף ריקווסטים בשנייה להביא פודים. אלף פודים בשנייה גם אם אנחנו משתמשים בפרוביז'ן קונטרול פלן אנחנו מגיעים ללימיטים.

התהליך: אדמין מגדיר פעם אחת SandboxTemplate, ואז SandboxWarmPool שמפנה אליו ומבקש מספר רפליקות חמות (בדוגמה — חמש). המשתמש מבצע SandboxClaim, וה-controller לא יוצר פוד חדש דרך ה-control plane אלא מחליף את השם של פוד חם קיים בשם שהתבקש, ומיד מחמם פוד נוסף כדי לשמור על גודל הבריכה. מעל זה אפשר לבנות gateway שמבקש סביבה מתוך הקוד: "אנחנו יכולים לפנות לגטווי הזה ולבקש סביבת סיינדבוקס תוך פחות משנייה" [18:39] — ואז להריץ פקודות, להעלות ולהוריד קבצים, וב-terminate הפוד חוזר לבריכה.

9. ארכיטקטורת inference טיפוסית על EKS

אג'נטים צורכים מודלים, ולא תמיד רוצים לצרוך אותם מ-Bedrock — בין אם כדי לבחון מודל חדש, להוריד עלויות, או להריץ מודל שעבר fine-tuning עצמי. ארז פרס את הארכיטקטורה שהוא רואה אצל לקוחות.



השלב הראשון בארכיטקטורה: הקצאת קומפיוט GPU בתוך הקלאסטר, שעליו ירוצו הפודים של המודל.

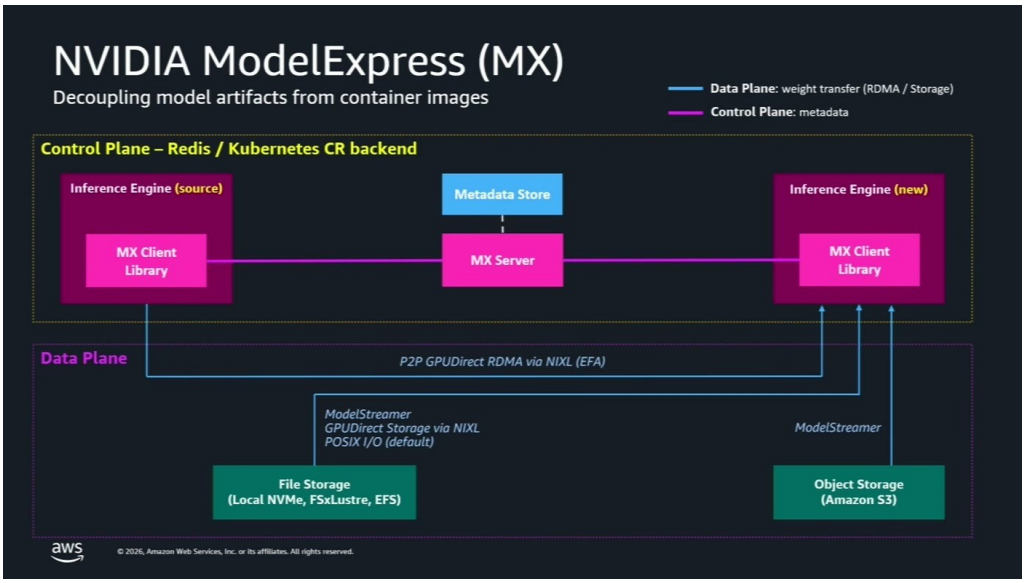
- **מנוע ה-inference.** הפופולריים היום — vLLM, SGLang ו-TensorRT — רצים כפודים על ה-GPU, והאימג' שלהם נשמר ב-ECR [20:09].
- **מקור המודל.** הדפוס הרווח הוא סנכרון מ-Hugging Face אל הסביבה, גם כדי לחסוך עלויות רשת וגם כדי להימנע מ-throttling.
- **אחסון המשקולות.** EFS, FSx (בעיקר FSx for Lustre למודלים גדולים), ו-Amazon S3.
- **Auto scaling ומדידה.** סקיילינג על מטריקות באמצעות KEDA או HPA.
- **חשיפת ה-endpoint.** API Gateway או NLB/ALB, לפי היכולות שרוצים לחשוף.
- **ניטור.** Amazon CloudWatch או השירות המנוהל של Prometheus.

10. האצת טעינת מודלים

האתגר הראשון שלקוחות פוגשים הוא בדיוק זה שיובל תתאר בהמשך מהשטח: node חדש עולה, וצריך למשוך אליו את האימג'. הבסיס כבר כבד — אימג' מינימלי של vLLM או SGLang שוקל 8 עד 10 ג'יגהבייט, כי הוא מכיל את המנוע, את Torch ולפעמים את CUDA. ואז מגיע הפיתוי לצרוב את המודל פנימה:

> **— ארז צרום, [21:42]** ראינו לקוחות שמריצים Container Imaging בין עשרות למאות ג'יגה. אז אל תעשו את זה.

החלופה היא להפריד את המודל מהאימג' ולהאיץ את שניהם בנפרד. הכלי הראשון הוא Run:ai Model Streamer — פרויקט אופן-סורס מבית Run:ai, שנרכשה על ידי NVIDIA. הוא יודע לעבוד מדיסק, מ-shared storage, וגם ישירות מול object storage כמו S3: כמה threads מושכים מ-S3 ב-Range Requests במקביל אל זיכרון ה-CPU, ובמקביל threads נוספים טוענים משם אל זיכרון ה-GPU. לפי ארז, S3 מגיע בדרך הזאת ל-throughput של 20–30 ג'יגהבייט לשנייה [21:42].



NVIDIA ModelExpress: control plane מבוסס Redis או Kubernetes CR מול data plane שמעביר משקולות ב-RDMA, עם fallback ל-file storage ול-object storage.

הכלי השני, חדש יותר, הוא ModelExpress תחת פרויקט NVIDIA Dynamo. בלב שלו עומד ModelExpress Server עם מסד נתונים (Redis, או ה-CR של Kubernetes עצמו) ששומר מטא-דאטה על ה-workers. worker שעולה שואל את השרת אם מישהו כבר מחזיק את המודל הזה בזיכרון. אם לא — נעשה fallback למסלול הרגיל: Run:ai Model Streamer מ-S3, או טעינה מהדיסק.

המספרים שהוצגו נמדדו על GLM שוקל כ-700 ג'יגהבייט:

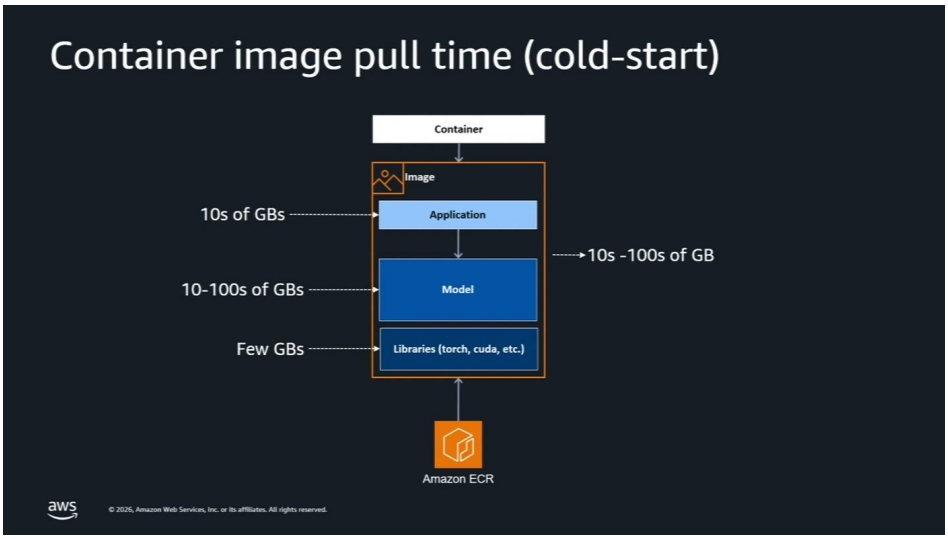
> **— ארז צרום, [23:12]** אני אתן דוגמה, מהבדיקות האחרונות שעשינו על GLM53 אנחנו מגיעים לאזור המאה שניות למודל של 700 ג'יגה בערך

זה ה-worker הראשון, ה-source. ה-worker השני הוא הסיפור האמיתי: על אינסטנסים מסוג P או G7E בגודל 12xlarge ומעלה אפשר לטעון את המשקולות דרך RDMA מעל EFA ישירות מה-source — "כמה מהירה? אזור ה-30 שניות למשקולות של 700 ג'יגה" [24:44].

אבל טעינת המשקולות היא רק חלק מהזמן. ארז ציין ש-70% בערך מזמן ההתנעה מתבזבז על compilation cache, deep GEMM וקרנלים שונים [23:12]. תוספת חדשה מעבירה גם את ארטיפקטי ה-compilation מה-source ל-target, וההשפעה על end-to-end דרמטית: עבור אותו GLM, מכ-14 דקות עד שהמודל מוכן להגשה — לאזור "השתי דקות ו-48 שניות. שזה ממש ממש משמעותי" [24:44].

11. SOCI: להאיץ את הקונטיינר עצמו

הפרדנו את המודל — נשאר הקונטיינר. SOCI הוא snapshotter של containerd, ולפי ארז הוא תומך בכל registry פופולרי שתומך ב-Range Requests — "תכלס עד עכשיו לא פגשתי רג'יסטרי שלא תומך בזה" [24:44], בין אם Docker, JFrog או Harbor.



מבנה זמן ה-cold start: שכבת האפליקציה בעשרות ג'יגהבייט, המודל ב-10 עד מאות ג'יגהבייט, והספריות (Torch, CUDA) בכמה ג'יגהבייט.

התהליך: פוד עולה, containerd מבצע pull ומוריד תחילה את ה-manifest, שמכיל בין היתר את רשימת ה-blobs — שכבות הקונטיינר, שב-ECR נשמרות ב-S3. SOCI פונה ל-registry ומוריד כל שכבה במקביל עם Range Requests, וכך משיג מקביליות שלא הייתה קיימת קודם.

החלק שרבים מפספסים הוא הצד השני. ההורדה מהירה, אבל כ-70% מהזמן הולך על decompression, ועד לא מזמן הוא היה סדרתי. עם SOCI הוא מקבילי, כך שאפשר לנצל את ה-CPU. ארז ציין שב-benchmark שביצע גילה שדורות CPU שונים נותנים ביצועים שונים, ושנוספו חבילות שמנצלות את ה-instruction set של ה-CPU לשיפור נוסף ב-decompression [26:16]. התוצאה:

> **— ארז צרום, [27:46]** לקבל את הקונטיינר ממה שהיה פעם כמה דקות, ל-30-40 שניות לקונטיינר של VLLM, למשל, של 10 ג'יגה.

12. Kaltura: אבטארים שמגיבים בזמן אמת

יובל רבובסקי מ-Kaltura לקחה את כל מה שנאמר עד כה והראתה אותו בפרודקשן. Kaltura חוגגת השנה 20 שנה, והתחילה כחברת מדיה — VOD, וובינרים. השאלה שהובילה לשינוי, בניסוח שלה: מה יקרה אם לא רק ננהל וידאו אלא ניצור וידאו באמצעות AI, ומעבר לזה — ניצור וידאו שמגיב בלייב למה שקורה במציאות. לשם כך נרכש בין היתר סטארט-אפ ישראלי בשם e-Self AI, שעושה בדיוק את זה [28:31].

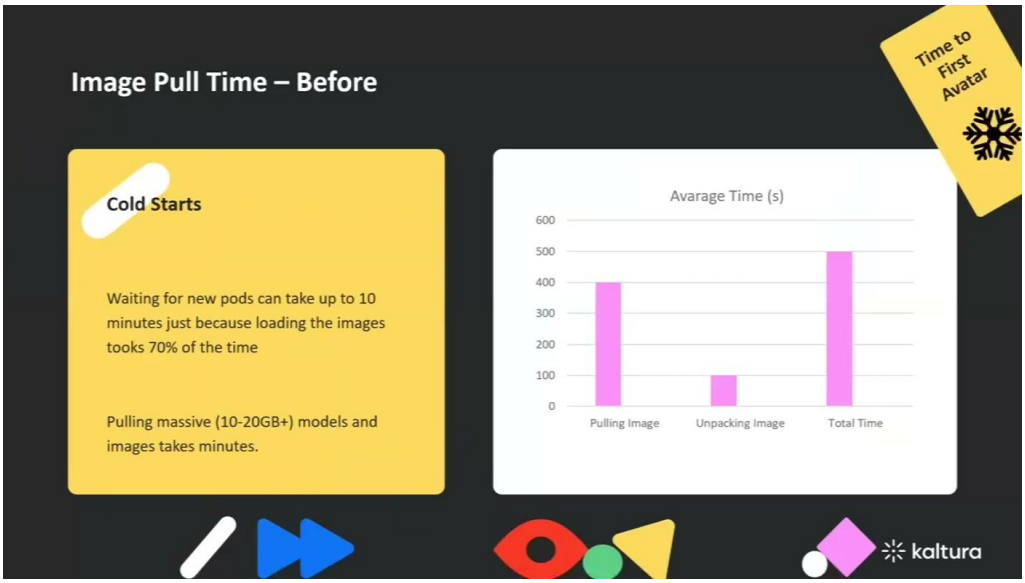
על הבמה רצה דמו חי: יובל שוחח עם האבטאר שלה, ביקשה הסבר על משקולות של מודל, ביקשה את אותו הסבר בספרדית (וקיבלה), ולבסוף ביקשה גרף השוואת מחירים — והאבטאר השיב "חכי רק רגע, אני מכינה לך גרף" [29:17].

- **ASR — Automatic Speech Recognition.** ממיר את הדיבור של המשתמש לטקסט, שנשלח ל-LLM.
- **TTS — Text to Speech.** ממיר את תשובת ה-LLM מטקסט לקטע קול.
- **STV — Speech to Video.** ממיר את קטע הקול לווידאו של האבטאר, כדי שאפשר יהיה גם לראות אותו ולא רק לשמוע [30:49].

שלושת המודלים עבדו ב-real time ועברו טסטים; הצוות העלה אותם ל-Kubernetes, הגדיר auto scaling, וזה נראה מוכן לפרודקשן. ואז נפגשו עם העולם האמיתי.

13. הבעיה הראשונה: cold start של 12 דקות

> **— יובל רבובסקי, [30:49]** יש ספייק, עולים נודים חדשים כל הפודים באמת סקייג'ולד על אותם נודים אבל הפודים עצמם פשוט לא עולים אנחנו מחכים 8 דקות, 9, 10, 12 דקות 12 דקות שאנחנו רק מחכים בשביל שפודים שכשהם כבר באוויר הם עונים ברילטיים רק בשביל שהם יעלו



המדידה שלפני: משיכת האימג' כ-400 שניות, פריסתו כ-100, וסה"כ כ-500 שניות — 70% מהזמן בטעינת אימג'ים של 10–20 ג'יגהבייט ומעלה.

יובל תיארה את הסתירה בין שני כוחות: מצד אחד רוצים שהאבטארים יעלו מהר ויעמדו בסקייל; מצד שני לא רוצים להחזיק עשרת אלפים אבטארים באוויר כשלא צריך אותם, כי זה יקר [30:49].

המחקר הצביע על cold start, וספציפית על זמן משיכת האימג'. הצעד הראשון היה ההיגיינה הרגילה — הוצאת המודלים מתוך האימג', הורדת תלויות, multi-stage builds — "ובאיזשהו שלב פשוט הבנו שלא משנה מה אנחנו נעשה האימג'ים האלה עדיין נשארו יחסית גדולים" [32:23]. כאן נכנס SOCI.

> **— יובל רבובסקי, [32:23]** ברגע שבעצם אפשרנו את סוצ'י ראינו ירידה משמעותית בעצם בזמני המשיכה של האימג'ים ממי באמת 8, 9, 10 דקות של משיכה לממש דקה 2 והדבר הכי טוב בכל הסיפור הזה זה שזה היה ממש פשוט להגדרה

ההפעלה עצמה הייתה בלוק קטן ב-user data: מכיוון שהם עובדים עם EKS ועם Karpenter, וה-AMIs שלהם הם Bottlerocket, כל מה שנדרש היה להגדיר ל-Bottlerocket להשתמש ב-SOCI, כמה שכבות למשוך במקביל, ומה גודל ה-chunks [32:23].

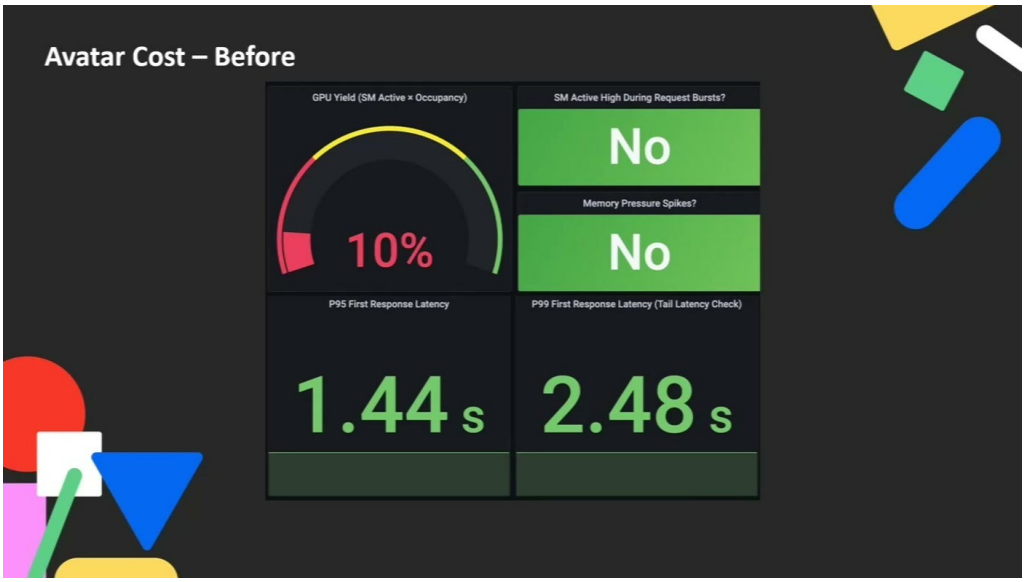
אבל זמן ההתנעה עדיין לא היה מספיק טוב — כי המשקולות, שהוצאו מהאימג', יושבות עכשיו ב-S3 ונטענות בצורה טורית: מ-S3 ל-CPU, ומה-CPU ל-GPU. הפתרון היה Run:ai Model Streamer, שהפך את כל השרשרת למקבילית — ה-CPU קורא מ-S3 בזמן שה-GPU קורא מה-CPU:

> **— יובל רבובסקי, [34:01]** וככה באמת הקטענו את זמן המשיכה של המשקולות שלנו את זמן התיינה של המשקולות שלנו לתוך המודלים ב-86%

14. הבעיה השנייה: 10% ניצולת GPU

הפודים עולים מהר, אבל העלות נשארה. יובל הדגישה את העיקרון: GPU הוא לא CPU, וצריך להתייחס אליהם כשני דברים שונים. Kubernetes יודע לחלק CPU בין פודים באופן נייטיב; ב-GPU אין מנגנון כזה. אם פוד משתמש ב-50% מה-GPU, ה-50% הנותרים פשוט יושבים idle ומשלמים עליהם [35:33].

הבעיה הגדולה יותר הייתה שאי אפשר היה לראות את הבעיה הראשונה. הניטור הראה ש-CPU תקין ושהזיכרון תקין, ולא רמז על צורך ב-right sizing. nvidia-smi הוא כלי CLI נחמד אך לא נתן מספיק הבנה; רק אחרי הוספת DCGM exporter ומטריקות SM Active הם ראו את התמונה האמיתית.

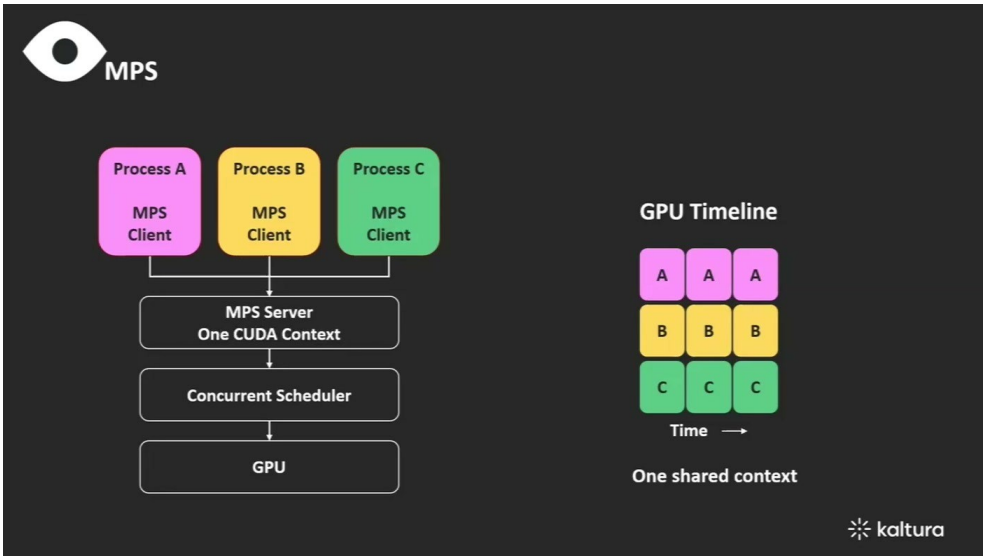


הדשבורד שחשף את הבזבוז: GPU Yield של 10%, בלי לחץ זיכרון ובלי SM Active גבוה בפיקים, ולצידם P95 של 1.44 שניות ו-P99 של 2.48 שניות.

> **— יובל רבובסקי, [37:04]** אנחנו מבזבזים מלא מלא כסף. מה זה מלא כסף? האחוז יוטיליזיישן שלנו עמד בממוצע על עשרה אחוזים שימוש בג'י פיו. זה אומר שפשוט יש לנו 90% מהג'י פיו שיושבים שם ולא עושים שום דבר ואנחנו משלמים עליהם.

מכאן נבחנו שלוש אסטרטגיות שיתוף GPU:

- **MIG — Multi-Instance GPU.** חלוקה קשיחה ברמת החומרה לאינסטנסים קטנים יותר. פתרון מצוין, אבל נתמך רק בסוגי אינסטנסים מאוד ספציפיים — שלא התאימו ל-[37:04] Kaltura.
- **Time slicing.** כל תהליך מקבל CUDA context משלו ופונה ל-time-sliced scheduler; ה-GPU מבצע context switching בין A, B ו-C לפי מקטעי זמן קבועים מראש.
- **MPS.** נוסף MPS Server באמצע; כל תהליך מקבל MPS Client, והשרת יוצא עם CUDA Context אחד — ולכן פונה ל-concurrent scheduler ולא ל-time-sliced, וה-GPU מטפל בכל ההקשרים במקביל ובאותו זמן [38:36].



MPS: שלושה תהליכים מול MPS Server יחיד עם CUDA context אחד, ולצידו ציר הזמן שבו A, B ו-C רצים באמת במקביל.

> **אזהרה מהדוברת** על time slicing ועל MPS גם יחד: "אין פה איזושהי הפרדה קשיחה של הזיכרון, וכן צריך לעשות הרבה בדיקות ולוודא שהוורקלודים שלנו לא מפריעים אחד לשני" [38:36]. בשונה מ-MIG, הבידוד כאן אינו חומרתי.

Kaltura בחרו ב-MPS, ושוב ההפעלה הייתה קצרה: CRD קטן של Node Overlay שמגדיר כמה תהליכים חולקים כל GPU, ו-Karpenter יודע לקחת את זה ולבצע [38:36]. התוצאה — "היום אנחנו עומדים בממוצע על 87% של יוטיליזיישן של כל GPU שלנו וזה באמת שיפור משמעותי של מעל פי 8" [38:36].

המסקנה האחרונה שלה היא הרחבת ההיקף: אף אחת מהאופטימיזציות אינה מותאמת ספציפית למודלים של Kaltura, ולכן אותה פלטפורמה מריצה היום גם אג'נטים — פנימיים, כמו NOC ו-FinOps agent שמקבלים התראות, מבצעים תחקור ומדווחים או פועלים בעצמם, וגם אג'נטים חיצוניים שנותנים support ללקוחות [40:07].

15. מה לוקחים מכאן

ארבעת ה-takeaways כפי שהוצגו: cold start, עלות, ביצועים, ואג'נטים על אותה פלטפורמה.

- **מדדו ניצולת GPU לפני שמדברים על עלות.** Kaltura גילו 10% ניצולת רק אחרי שהוסיפו DCGM exporter ומטריקות CPU. SM Active וזיכרון תקינים אינם עדות לניצולת GPU.
- **אל תצרבו מודלים לתוך container image.** "אז אל תעשו את זה" [21:42]. הפרידו את המשקולות ל-S3 או ל-FSx, והחזירו את זמן הטעינה עם Run:ai Model Streamer.
- **SOCI הוא ה-quick win עם היחס הטוב ביותר.** אצל Kaltura זה היה בלוק user data ב-Bottlerocket, והוריד משיכה של 8–10 דקות לכ-2 דקות.
- **שיתוף GPU הוא החלטת ארכיטקטורה, לא flag.** MIG נותן בידוד חומרתי אך מוגבל לאינסטנסים מסוימים; MPS ו-time slicing זמינים רחב יותר אך ללא הפרדת זיכרון — ודורשים בדיקות interference לפני פרודקשן.
- **לאג'נטים תכננו בידוד ברמת ה-runtime.** Kata Containers מעל Firecracker כ-RuntimeClass, Pod Identity ל-least privilege, ו-Network Policies ל-zero trust פנימי — לא קונטיינר רגיל.
- **בדקו את Agent Sandbox לפני שבונים scheduler משלכם.** זהות יציבה, suspend/resume, singleton ו-warm pool נותנים בדיוק את מה ש-Deployment ו-StatefulSet לא נותנים לאג'נט.
- **פלטפורמה אחת לכל עומסי ה-AI.** האופטימיזציות אינן ספציפיות למודל, ולכן אותה תשתית משרתת inference, אבטארים ואג'נטים כאחד.

בסיום הזמין דוב אמיר את הקהל לסרוק QR עם מדריכים, דוגמאות וארכיטקטורות end-to-end של AI על EKS, ולמלא את סקר הסשן [41:38].

מילון מונחים

מונח	מה זה	למה זה הופיע
DRA	Dynamic Resource Allocation — הקצאת מכשירים דינמית ב-Kubernetes מבוססת claims	מחליף device plugins; נתמך מ-EKS 1.33
SOCI	snapshotter של Seekable OCI — containerd שמקביל הורדת שכבות ו-decompression	הוריד cold start מדקות לעשרות שניות
MIG	Multi-Instance GPU — חלוקה קשיחה של GPU ברמת החומרה	נפסל ב-Kaltura בגלל תמיכה באינסטנסים מסוימים בלבד
MPS	Multi-Process Service — שרת שמאחד תהליכים ל-CUDA context אחד	הבחירה של Kaltura; 10% → 87% ניצולת
Time slicing	חלוקת GPU במקטעי זמן עם context switching	נבחן ונדחה לטובת MPS

מונח	מה זה	למה זה הופיע
Node Overlay	CRD של Karpenter להשפעה על scheduling ועל מאפייני משאבים	הכלי שבו הוגדר שיתוף ה-GPU בפועל
Agent Sandbox	פרויקט Kubernetes SIG Apps עם controller ו-CRDs לעומסי אג'נטים	זהות יציבה, singleton, warm pool
Strands Agents	SDK אופן-סורס של AWS לבניית אג'נטים ו-MCP servers	הדוגמה בקוד שהוצגה על הבמה
Kata Containers	runtime שמריץ כל פוד כמיקרו-VM (מעל Firecracker)	ה-best practice החדש לבידוד אג'נטים
Model Streamer	כלי של Run:ai לטעינה מקבילית של משקולות מ-S3 אל CPU ו-GPU	86% שיפור בזמן טעינת משקולות ב-Kaltura
ModelExpress	רכיב תחת NVIDIA Dynamo להעברת משקולות בין workers ב-RDMA	700GB ב-30 שניות בין source ל-target
DCGM exporter	יצואן מטריקות GPU של NVIDIA ל-Prometheus	הכלי שחשף את הבזבוז אצל Kaltura