Vision Models on AWS Lambda

Serving a heavy vision model on demand runs straight into a packaging problem that most model-serving writing skips, because most model-serving writing assumes a long-lived container with the model already resident in memory. AWS Lambda gives none of that for free: the deployment package is bounded, the layer is bounded, the runtime ships a deliberately minimal set of native tools, and the function is torn down between invocations. A raster-log digitiser whose trained checkpoint runs to hundreds of megabytes, whose dependency set is the full scientific-Python stack, and whose input is a single-channel grayscale scan up to 12,800 pixels wide, breaches every one of those limits at once. This whitepaper documents the packaging architecture we shipped to serve that model behind Lambda, and it makes one claim its central spine: the model cannot be served as a single deployment package, so it is decomposed into three artefacts, each of which exists to clear a different packaging constraint. The checkpoint is mounted from an Elastic File System, which removes it from the deployment-size accounting entirely and lets a function hydrate weights far larger than any package allows. The Python dependencies are published as a layer uploaded through S3, which keeps the function image small while still carrying a heavy import graph. An ImageMagick binary is shipped as its own layer, because the preprocessing that turns a raw grayscale scan into the single-channel tensor the model expects needs a native tool the Lambda runtime does not include. We set out the three limits explicitly, walk the decomposition that answers them, and describe the cold-start budget a batch-of-one request pays to hydrate the checkpoint from EFS and import the dependency layer, a cost paid only when a request actually arrives. Against that on-demand cost we place the fixed cost the posture avoids: an always-on GPU at 750 to 1800 EUR per month that bills whether idle or busy. The deliverable is a reusable packaging pattern for any team that has to serve a model too heavy for a function package, on infrastructure that only bills for the work it does.

Tarry Singh
Founder & CEO

Tannistha Maiti
Senior AI Researcher

Narendra Patwardhan
Research Collaborator

VEERNET / SERVERLESS / AWS-LAMBDA / MLOPS / MODEL-SERVING / EFS

CONTENTS

> "
>
> The checkpoint does not fit in the package. It was never going to fit in the package. The whole architecture follows from taking that sentence seriously instead of fighting it.
>
> "

I

THE PROBLEM

A heavy model breaches every serverless limit at once

What we were actually trying to ship

We had a trained segmentation model, VeerNet, that turns a scanned raster well log into a digitised curve. The geoscience behind it, the loss-function choices, and the training under a hard memory ceiling are the subjects of their own documents. This one is about a narrower and more stubborn problem that came after the model was already good: how to serve it so that an operator can send one scan and get one result back, without us keeping a machine running for them, and without paying for a GPU that sits idle between requests.

The obvious answer for on-demand, pay-per-use inference is a function platform, and AWS Lambda is the one we reached for. The premise is attractive. You pay for the milliseconds a request runs and nothing when no request is running, the platform handles the scaling, and there is no server for anyone to patch. For a workload that is bursty by nature, where an operator digitises a batch of scans one week and nothing the next, that billing shape is close to ideal.

The premise collides immediately with the model. A function platform is built for small, quick, mostly stateless code: parse an event, do a little work, return. Our model is none of those things. The trained checkpoint runs to hundreds of megabytes. The code that loads and runs it drags in the full scientific-Python stack. And the input is not a tidy JSON payload but a grayscale image that can be 12,800 pixels wide, which has to be decoded and normalised into a single-channel tensor before the network ever sees it. Each of those three facts hits a different hard limit of the platform, and it hits it at the same time as the other two.

That simultaneity is the whole difficulty. Any one of the three constraints has a well-worn workaround. All three at once forces a different way of thinking about what a deployment even is. The move that made the workload fit was to stop treating the model as one thing

you deploy and start treating it as three artefacts you assemble, each one shaped to clear exactly one of the limits.

THE THREE FACTS ABOUT THE MODEL, AND THE LIMIT EACH ONE BREACHES

hundreds of MB
Trained checkpoint size, past any deployment package

full sci-Python
Dependency graph, too heavy for the function image

1 channel
Grayscale input needing a native decode tool

12,800 px
batch of 1
Widest scan the decode has to handle

The rest of this document is the anatomy of that decomposition. We name the three limits, describe the artefact that answers each, walk the cold-start budget the assembled function pays on demand, and set that on-demand cost against the fixed cost the whole posture is chosen to avoid.

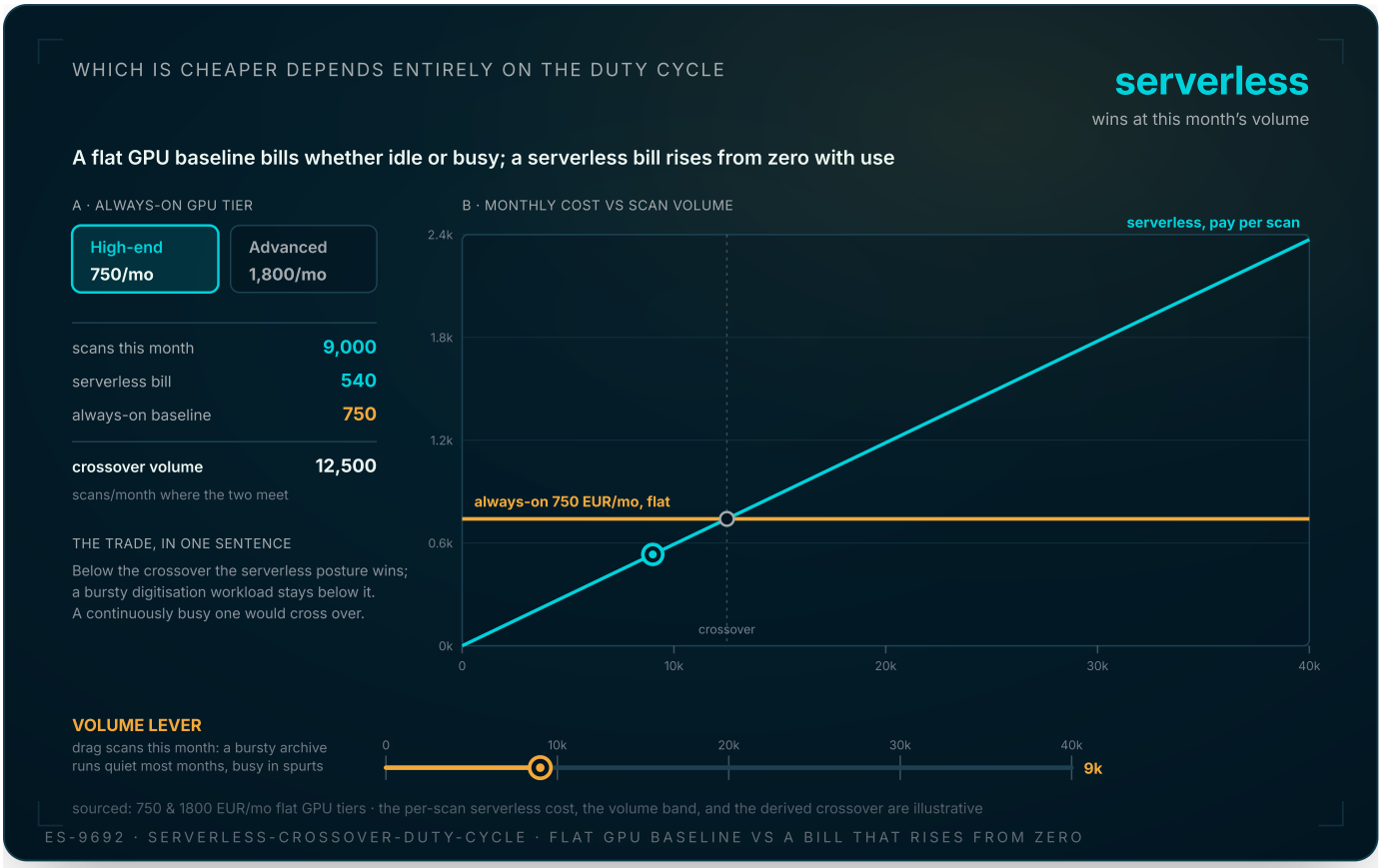
Why a long-lived server was the thing to avoid

Before the decomposition, it is worth being honest about the alternative, because the alternative is what makes the packaging work worth doing. The straightforward way to serve a heavy model is a long-lived server with the model loaded into memory once and

kept there. It sidesteps every packaging limit, because there is no function package to bound: you install what you like, mount what you like, and the checkpoint is a file on a disk you control.

The cost of that comfort is that the machine runs whether or not anyone is using it. For a model that needs a GPU to serve at a reasonable latency, the fixed cost is real. In our engagement the rentable GPU tiers were 750 EUR per month for a high-end card and 1800 EUR per month for an advanced one, and that number is charged in full for a month whether the card served ten thousand scans or none. For a workload that is quiet most of the time, you are paying continuously for capacity you use intermittently. The serverless posture inverts that: you pay for the seconds a request runs and nothing in between, at the cost of the packaging discipline this whitepaper is about, and the cold start each idle-to-busy transition pays. Which side of that trade is right depends entirely on the duty cycle, and for a bursty digitisation workload the on-demand side wins clearly enough to be worth the engineering.

The instrument below makes that duty-cycle trade adjustable. The always-on GPU baseline is flat at the tier you pick, and the serverless bill rises from zero with the scans you actually run, so the two lines cross at a break-even volume: below it the serverless posture is cheaper, and a bursty archive stays below it most months.



Whether serverless or an always-on GPU is cheaper is not a fixed answer; it depends entirely on the duty cycle. The always-on GPU baseline is flat, 750 EUR per month for a high-end card or 1,800 for an advanced one, and it bills that whether the machine serves ten thousand scans or none. The serverless bill starts at zero and rises with each scan, so the two lines cross at a break-even volume: below it the serverless posture is cheaper, above it the always-on machine is. Toggle the tier to set the height of the flat baseline, and drag the volume lever to place this month's scan count on the rising serverless line and read which side of the crossover it lands on. The orange element is the only one that argues: the flat baseline the serverless posture stays under precisely because a bursty digitisation archive runs quiet most months and busy only in spurts. The two GPU tier prices are sourced from the engagement archive; the per-scan serverless unit cost, the monthly volume band, and therefore the derived crossover volume are illustrative inputs, so this is the shape of an engineering trade, not a quoted price.

II

THE LIMITS

Three bounded resources, and none of them negotiable

The deployment-size limit, and why the checkpoint cannot travel in the package

A Lambda function is deployed from a package, and that package is size-bounded. This is a deliberate platform decision, not an oversight: the whole point of a function platform is that packages are small enough to distribute and start quickly, so the limit is part of the contract [2]. Nothing about it is going to move for our convenience.

The trained checkpoint is the first thing that does not fit. An encoder-decoder segmentation network with a residual encoder and an attention refinement on the bottleneck carries a lot of parameters, and serialised to disk the weights run to hundreds of megabytes [4][5]. You cannot put a file that large inside a package that is bounded well below it. You cannot compress your way out either, because trained weights are close to incompressible and the limit applies to the unpacked size anyway. This is not a tuning problem. The checkpoint and the package are simply different orders of magnitude, and the only real answer is to stop counting the checkpoint against the package at all.

The layer-weight limit, and why the dependency stack is not the code

The second limit is on layers, the separately packaged archives a function mounts alongside its own code. Layers are how you keep shared libraries out of every function image, and they too are size-bounded. That matters because the code that runs our model does not stand alone: it imports the numerical and image-handling stack that deep-learning inference depends on, and that import graph is heavy in its own right, independent of the checkpoint.

If you bundle that whole stack into the function image, the image itself becomes large and slow to distribute, and you pay that weight on every deploy. The dependencies are logically separate from the model code and separate from the checkpoint, and they want to be packaged separately so they can be versioned once and shared, rather than re-shipped every time the function code changes by a line. That separation is what a layer is for, and it is the second artefact in the decomposition.

The native-tooling gap, and why grayscale decode needs a binary

The third limit is subtler because it is a limit of omission rather than of size. The Lambda runtime ships a deliberately minimal set of native tools. That keeps the runtime small and its attack surface narrow, and for most function workloads it is exactly right. It is wrong for ours in one specific place: the preprocessing.

The model does not consume a raster file. It consumes a single-channel tensor of a particular shape and normalisation, and getting there from a raw grayscale scan is real image work: decode the file, collapse it to the one grayscale channel the network expects, and resize or pad it into the geometry the model was trained on, for scans that can be 12,800 pixels across. That work is done well by a mature native image tool, ImageMagick, and badly or not at all by whatever the bare runtime happens to include. The runtime does not ship it, so we have to. The native binary, packaged as its own layer, is the third artefact, and it answers a gap the other two do not touch.

III

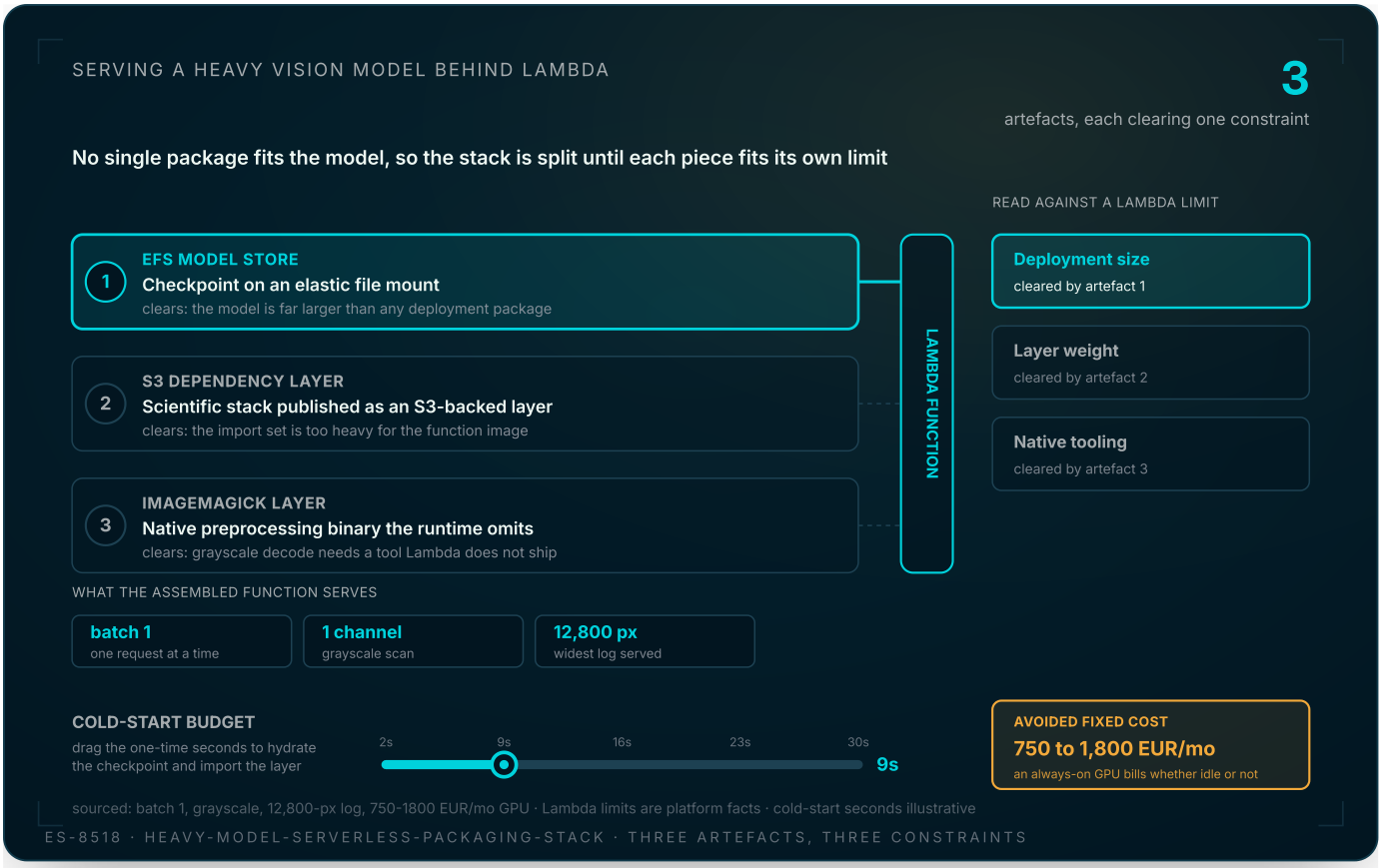
THE DECOMPOSITION

Three artefacts, each shaped to clear one limit

The stack, read as an argument

At this point the packaging architecture is forced. There are three hard limits and three facts about the model that breach them, and the clean solution is a one-to-one map: one artefact per limit, each shaped to the constraint it clears and to nothing else. The checkpoint moves to an EFS mount to clear the deployment-size limit. The dependencies move to an S3-backed layer to clear the layer-weight limit. The preprocessing tool moves to an ImageMagick layer to clear the native-tooling gap. The function itself becomes small: it is the code that assembles these three, receives a request, and returns a result.

The instrument below is that argument in one picture. Each lane is an artefact, each artefact names the constraint it clears, and the toggle reads the stack against each of the three Lambda limits in turn so you can see that every limit has exactly one artefact answering it. The cold-start bar shows the one-time seconds a batch-of-one grayscale request pays to hydrate, and the single orange element carries the economic point the whole architecture is in service of.



How a multi-hundred-megabyte segmentation model is served behind AWS Lambda: not as one deployment package, which no single package could hold, but as three artefacts that each clear a different packaging constraint. Artefact one is the model checkpoint on an EFS mount, which answers the deployment-size limit because the weights are far larger than any function image or layer. Artefact two is the scientific dependency stack published as an S3-backed layer, which answers the layer-weight limit and keeps the function image itself small. Artefact three is an ImageMagick binary layer, which answers the native-tooling gap because turning a raw grayscale scan into the single-channel tensor the model expects needs a binary the Lambda runtime does not ship. The toggle on the right reads the stack against each Lambda hard limit and highlights the one artefact that answers it. The cold-start budget bar shows the one-time seconds a batch-of-one grayscale request pays to hydrate the checkpoint and import the layer, paid only when a request actually arrives. The single orange element carries the argument: the always-on GPU baseline of 750 to 1800 EUR per month that this posture avoids, a fixed cost that bills whether or not any request ever comes. The batch size, grayscale channel, widest-log width, and GPU baseline are sourced from the engagement archive; the Lambda platform limits are AWS-documented platform facts; the cold-start seconds are an illustrative input.

Read the lanes top to bottom and the decomposition is legible as three independent decisions that happen to compose. The EFS lane clears deployment size because the weights never enter the package. The S3-layer lane clears layer weight because the heavy imports live in a versioned, shared archive rather than in the function image. The ImageMagick lane clears the native gap because the decode tool travels as a binary the

runtime lacks. Toggle between the limits and the highlighted lane changes, which is the visual form of the claim: no limit is left unanswered, and no artefact is doing two jobs at once.

> **THE POSTURE THE DECOMPOSITION ENCODES**
>
> Do not try to make a heavy model fit inside a function package. It will not, and the effort spent trying is wasted. Instead, list the platform limits the model breaches, and for each limit choose the artefact and the storage that removes the model's mass from that particular accounting. The function that remains should be small enough to feel like an ordinary function, because all its weight has been pushed into artefacts that are mounted rather than packaged.

Artefact one: the checkpoint on an EFS mount

The checkpoint is the heaviest single thing and the one that most obviously cannot travel in the package, so it moves to an Elastic File System mount. The function is configured with the file system attached, and at cold start it reads the weights from the mount into memory rather than carrying them in its own image. The effect on the packaging accounting is total: the checkpoint contributes nothing to the deployment package, because it is not in the deployment package. It is a file on a mounted disk, and the function's relationship to it is the same as any process reading a large file it did not ship with.

There is a real cost, and it is the read at cold start. Hydrating hundreds of megabytes of weights from a mount into memory takes time, and that time is part of the cold-start budget the next section treats explicitly. But it is a one-time cost per cold function, not a per-request cost, and crucially it is a cost paid only when a request arrives to warm the function in the first place. An idle function costs nothing, checkpoint and all, because the checkpoint is sitting on a mount that is only read when something reads it.

Artefact two: the dependency stack as an S3-backed layer

The scientific-Python dependencies move into a layer, and because the set is large the practical path to publish it is through S3: the layer archive is uploaded to S3 and the function references it from there. Two properties make this the right shape. First, the function image stays small, because the heavy imports are in the mounted layer rather than the image, so deploying a change to the function code does not re-ship the whole stack. Second, the layer is versioned and shareable, so the dependency set is built and validated once and then referenced, rather than rebuilt on every function deploy. The dependencies are genuinely a separate concern from the model code and from the weights, and giving them their own artefact makes that separation concrete rather than notional.

Artefact three: ImageMagick as a native layer

The preprocessing tool is the last artefact and the one that is easiest to forget until a scan arrives and there is nothing to decode it with. The single grayscale channel and the 12,800-pixel width are not incidental details; they are the reason a generic, pure-Python decode is not enough and a mature native tool is. ImageMagick is packaged as its own layer, separate from the Python dependency layer, because it is a different kind of thing: a compiled binary rather than an importable library, answering a gap in the runtime rather than adding to the import graph. Keeping it as its own layer keeps each layer coherent, one for the interpreted stack and one for the native tool, and keeps the function image free of both.

> **“The function ended up being the smallest part of the whole system. Everything heavy had been moved into something mounted, so the code that was left just orchestrated the assembly.”**
>
> — FROM OUR OWN DEPLOYMENT NOTES

IV

THE BUDGET

Cold start is a real cost, and it is the right one to pay

What a request actually pays

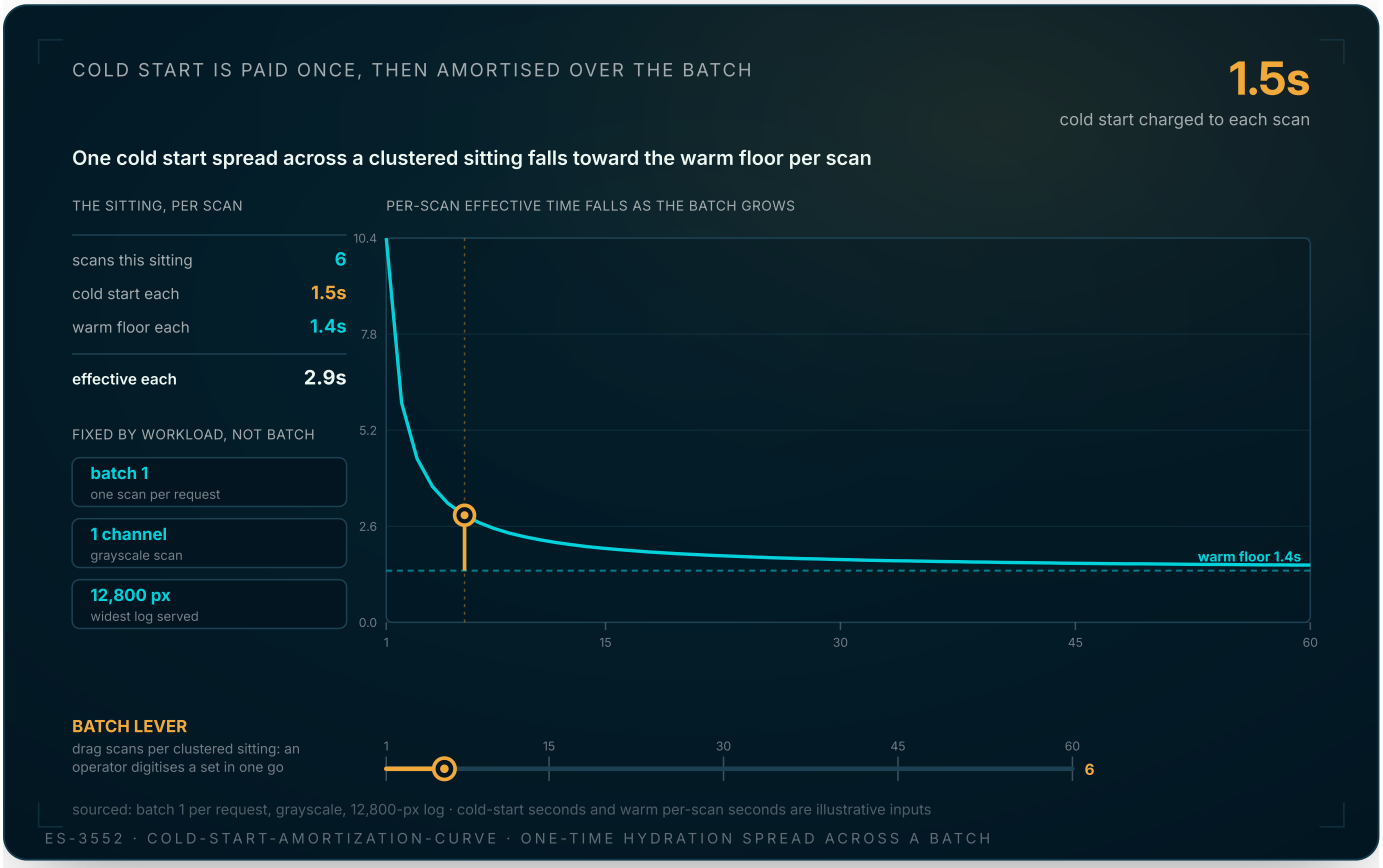
An assembled function is not free to start. The first request to a cold function pays a cold-start cost that is the sum of the platform acquiring a runtime, mounting the layers and the file system, importing the heavy dependency graph, and reading the checkpoint from the EFS mount into memory [3]. Those last two are the ones our decomposition adds: a large import graph takes time to import, and hundreds of megabytes of weights take time to read from a mount. This is not hidden and it should not be. It is the price of not keeping a machine warm, and the honest way to reason about it is as a budget rather than as a surprise.

The cold-start bar on the instrument is that budget made adjustable. It shows the one-time seconds a batch-of-one grayscale request pays to hydrate, and it is worth internalising three things about that number. It is one-time per cold function, not per request; a warm function serves subsequent requests without repaying it. It is paid only when a request arrives, so an idle service that no one is using pays nothing at all. And it is a latency cost, not a fixed monetary cost, which is exactly the trade the posture is built around: accept a first-request delay in exchange for paying nothing when there is no first request.

The mechanics of that per-function lifecycle are worth being precise about, because they decide how often the budget is actually paid. When a request arrives and no warm function instance exists, the platform provisions one: it acquires a runtime, attaches the layers and the file system, runs the initialisation code, and only then handles the request. That initialisation is where our two heavy costs land, the import of the dependency layer and the read of the checkpoint from the mount. Once initialised, the instance stays warm for a while and serves further requests directly, so a burst of scans from one operator pays the cold start once at the front and then runs warm through the rest of the batch. It is when traffic goes quiet long enough for the instance to be reclaimed, and then a new request arrives,

that the budget is paid again. This is why cold start is a duty-cycle cost rather than a per-request one: the more clustered the requests, the fewer cold starts per unit of work, and the digitisation workload is naturally clustered, because an operator processes a set of scans in a sitting rather than one every few hours.

The instrument below is that amortisation made adjustable. Drag the number of scans in one sitting and watch the per-scan cold-start overhead, the one-time hydration spread across the batch, collapse toward the warm-inference floor: a single-scan sitting pays the whole cold start once, while a large sitting serves at close to warm latency.



Why cold start is cheap for a bursty workload: it is a one-time, per-warm-instance tax, so a clustered sitting of scans amortises it toward zero per scan. A cold function pays the hydration once at the front of the batch, and every scan after the first reuses the resident checkpoint and runs warm. Drag the batch lever to set how many scans an operator digitises in one sitting: one cold start is spread across the whole batch, so the per-scan cold-start overhead, the orange gap above the warm floor, collapses as the batch grows. A single-scan sitting pays the whole cold start once; a large sitting serves at close to warm latency. The teal curve is the effective per-scan time, the amortised cold start plus the warm inference floor it descends toward. The batch-of-one request shape, the single grayscale channel, and the 12,800-pixel widest served log are sourced from the engagement archive; the cold-start seconds and the warm per-scan inference seconds are illustrative inputs whose magnitudes a team should measure for its own deployment, though the shape, one-time versus per-scan, is real.

There is a design consequence in that lifecycle worth stating. Because the checkpoint read happens in initialisation and not per request, the way to keep the served latency low is to make the initialisation do as much as it can once and reuse it. The model is loaded into memory in the initialisation and held there for the life of the warm instance, so every request after the first reuses the resident weights rather than re-reading the mount. The preprocessing binary from the ImageMagick layer is present from the moment the instance exists, so it too is a one-time attachment rather than a per-request fetch. The decomposition therefore does more than make the model fit; it also cleanly separates the costs that belong in initialisation, the dependency import and the checkpoint read, from the cost that belongs per request, which is the inference itself over one scan.

The cost this posture avoids, which is the point of all of it

The reason to accept a cold-start budget at all is the orange element on the instrument, and it is the single fact the whole architecture argues toward. A long-lived inference server sized for this model needs a GPU, and that GPU bills continuously at 750 to 1800 EUR per month whether it is serving requests or sitting idle overnight and all weekend. That is a fixed cost that does not care about your duty cycle. The serverless posture converts that fixed monthly cost into a variable per-request cost plus a cold-start latency tax, and for a workload that is busy in bursts and quiet the rest of the time, that conversion is heavily in your favour. You pay the packaging complexity and the cold start once each, in engineering and in latency, and in exchange you stop paying for idle GPU capacity every single month.

This is why the avoided baseline, not the cold-start seconds, is the element drawn in the scarce accent colour. The cold start is a cost you manage; the avoided baseline is the reason the whole exercise pays for itself. A team weighing this architecture should read the two together: the on-demand cost is real but bounded and only incurred on use, and the fixed cost it displaces is real and incurred always.

V

THE METHOD

Reproducing the packaging pattern

The pattern, stated so it transfers

The architecture generalises past our specific model, and the generalisation is worth stating plainly because it is the reusable part. When you have to serve a model that is too heavy for a function package, do not fight the package limit. Instead, enumerate the platform limits the model breaches, and for each one choose the artefact and the storage that removes the model's mass from that particular accounting.

For a heavy checkpoint, mount it from a file system attached to the function so it never enters the package, and accept the read into memory as a cold-start cost. For a heavy dependency graph, publish it as a versioned layer, uploaded through S3 when the set is large, so the function image stays small and the stack is shared rather than re-shipped. For a preprocessing step that needs a tool the runtime omits, ship that tool as its own native layer, kept separate from the interpreted-dependency layer so each artefact stays coherent. What remains, the function itself, should be small: the orchestration that receives a request, runs the preprocessing through the native tool, hydrates or reuses the model from the mount, runs inference, and returns the result.

The discipline that makes this trustworthy rather than hopeful is to treat the cold-start budget as a first-class number you measure and defend, not a detail you discover in production. Know how long the checkpoint read takes, know how long the import graph takes, and decide deliberately whether the resulting first-request latency is acceptable for the workload or whether it needs a warming strategy. And keep the accounting explicit: every heavy thing about the model should be traceable to the artefact that carries it and the limit that artefact clears, so that when the model grows or a new limit appears, you know exactly which artefact has to change.

Checkpoint on EFS, not in the package

- The trained weights run to hundreds of megabytes, past any deployment package
- An Elastic File System mount removes the checkpoint from the package accounting entirely
- The function reads the weights from the mount at cold start rather than carrying them
- This is the artefact that answers the deployment-size limit

Dependencies as an S3-backed layer

- The scientific-Python import graph is too heavy to bundle into the function image
- Publishing it as a layer uploaded through S3 keeps the function image small
- The layer is versioned and shared, not re-uploaded on every function deploy
- This is the artefact that answers the layer-weight limit

ImageMagick as its own layer

- A raw grayscale scan has to be decoded and normalised before the model sees it
- That preprocessing needs a native binary the Lambda runtime does not ship
- Shipping ImageMagick as a layer supplies the tool without inflating the image
- This is the artefact that answers the native-tooling gap

WHAT TO CARRY OUT OF THIS

1. A multi-hundred-megabyte model cannot be served as a single Lambda deployment package, so it is decomposed into **three** artefacts, each of which clears a **different** packaging constraint rather than one artefact trying to clear all of them.
2. The checkpoint lives on an **EFS** mount and is read at cold start, which removes it from the deployment-size accounting entirely, because the weights are far larger than any function package allows.
3. The scientific-Python dependency stack ships as an **S3**-backed layer, which keeps the function image small and lets the heavy import graph be versioned and shared instead of re-shipped on every deploy.
4. An **ImageMagick** layer answers the native-tooling gap, because turning a raw grayscale scan up to **12,800** pixels wide into the single-channel tensor the model expects needs a compiled binary the runtime does not ship.
5. The cold start is a one-time, on-demand latency cost paid only when a request arrives; against it the posture avoids an always-on GPU baseline of **750** to **1800** EUR per month that would bill whether idle or busy.

Limitations

The cold-start seconds shown on the budget bar are illustrative inputs, not measured latencies from a specific deployment; the shape of the cost, a one-time hydration paid only when a request arrives, is real, but the exact number depends on the checkpoint size, the memory the function is allocated, the file-system throughput, and the import graph, and a team adopting this pattern should measure its own budget rather than adopt ours. The specific platform limits that force the decomposition, a bounded deployment package, a bounded layer, and a minimal native runtime, are AWS platform facts that can and do change over time and across configurations, so the enumeration matters more than any single threshold: the method is to list the limits your model breaches on the platform version you actually target and answer each one, not to memorise a limit that may move. The GPU baseline of 750 to 1800 EUR per month is the sourced fixed cost from the engagement and is the cost the posture avoids, but whether avoiding it is worth the packaging complexity and the cold-start latency depends entirely on the duty cycle: a workload that is busy

continuously would be better served by the long-lived server this architecture is chosen against, and the serverless posture wins specifically because our digitisation workload is bursty. Finally, this whitepaper is about packaging and serving, not about model accuracy; the quality of the digitised curve is set by the training and the architecture and is documented separately, and nothing in the packaging changes what the model predicts, only where its mass lives and when its cost is paid.

References

1. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., Dennison, D. (2015). *Hidden Technical Debt in Machine Learning Systems.* NeurIPS. The reference account of why the trained model is a small box in a large system, and why the serving and configuration code around it dominates the real cost. https://papers.nips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html

2. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., Patterson, D. A. (2019). *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* UC Berkeley technical report. The systems account of what serverless does and does not give you, including statelessness, limited local resources, and the cold-start tax. https://arxiv.org/abs/1902.03383

3. Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M. (2018). *Peeking Behind the Curtains of Serverless Platforms.* USENIX ATC. The measurement study of how function platforms actually behave, including cold-start latency and resource limits. https://www.usenix.org/conference/atc18/presentation/wang-liang

4. Ronneberger, O., Fischer, P., Brox, T. (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation.* MICCAI. The encoder-decoder shape the served model belongs to, whose parameter count is what makes the checkpoint too large for a function package. https://arxiv.org/abs/1505.04597

5. He, K., Zhang, X., Ren, S., Sun, J. (2016). *Deep Residual Learning for Image Recognition.* CVPR. The residual encoder the served network is built on, included for the architecture that sizes the checkpoint the EFS mount has to hold. https://arxiv.org/abs/1512.03385

6. Koroteev, D., Tekic, Z. (2021). *Artificial intelligence in oil and gas upstream: Trends, challenges, and scenarios for the future.* Energy and AI. Context for why an operator with a legacy raster-log archive would want an on-demand digitiser it does not have to keep a GPU running to use. https://www.sciencedirect.com/science/article/pii/S2666546820300033

Get the full whitepaper

This page is the long-form summary. The complete whitepaper adds the per-artefact packaging manifest, the exact cold-start breakdown between the dependency import and the checkpoint read, the function orchestration that assembles the three artefacts on each invocation, and the duty-cycle analysis that decides when the serverless posture beats the always-on GPU baseline.

EARTHSCAN

Serverless Inference for Heavy Vision Models on AWS Lambda

Published October 2022. Generated 2026-07-22.
https://www.earthscan.io/whitepapers/serverless-inference-for-heavy-vision-models-on-aws-lambda
Copyright EarthScan. All rights reserved.