# Comparing AWS Amplify vs AWS ECS Fargate for a Mobile App Backend

## Introduction

In planning **Version 2** of a mobile app with thousands of users, we consider two AWS-based backend approaches: **AWS Amplify** and **AWS ECS Fargate** (provisioned via Terraform). The app’s backend must handle user-generated video uploads (with processing) and user comments, while scaling to thousands of active users. Key requirements include: secure user authentication, robust storage for videos and comments, high scalability, and an architecture that is cost-efficient during development and easy for a remote team to maintain. This report compares Amplify’s managed, serverless approach against an ECS Fargate containerized microservices approach on the metrics of cost, performance & scalability, developer experience, operations overhead, and deployment speed. A summary comparison table and a recommendation are provided.

## Architecture Requirements and Context

The backend for this mobile app must support:

* **Video Upload, Storage & Processing:** Users will upload video files; the system should store these (e.g. in S3) and process them (e.g. transcode or generate thumbnails).
* **Comment Storage & Retrieval:** Users can post comments on videos, requiring a database or storage solution for comments and an API to retrieve them.
* **Scalability:** The platform should seamlessly scale to handle **thousands of active users** and variable load (potentially sudden spikes in traffic or uploads).
* **Secure Authentication & Access Control:** User sign-up/login and data access must be secure (likely via robust services like AWS Cognito for authentication tokens).
* **Remote Team Collaboration:** A distributed development team should be able to work on the backend with minimal friction – e.g. easily deploying their changes, having separate dev/test environments, and using Infrastructure as Code or similar for consistency.
* **Low Operational Overhead:** The team prefers to focus on application code and features rather than managing servers or complex infrastructure, especially during ongoing development.
* **Cost Optimization (Dev & Scale):** Keep costs low during development (idle or low-traffic periods) and ensure the solution remains cost-effective at scale (when user activity and resource consumption are high).

With these needs in mind, we evaluate **AWS Amplify**, a managed backend-as-a-service platform, versus **AWS ECS Fargate**, a container orchestration service, for implementing the backend.

## AWS Amplify Overview for this Use Case

AWS Amplify is a **full-stack development platform** that streamlines building web and mobile applications by integrating a suite of AWS managed services[[1]](https://bobcares.com/blog/amplify-aws-fargate/#:~:text=On%20the%20other%20hand%2C%20AWS,for%20building%20and%20deploying%20applications)[[2]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=AWS%20Amplify%20is%20a%20solution,the%20fiercely%20competitive%20digital%20landscape). Amplify essentially abstracts away much of the traditional infrastructure setup, allowing developers to focus on features instead of server management[[3]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=developers%20required%20a%20platform%20that,faster%2C%20iterate%20on%20concepts%20more). It provides a CLI and libraries to easily add capabilities like authentication, APIs, storage, and hosting. For our video app scenario, Amplify offers the following relevant features:

* **Managed Storage & File Handling:** Amplify’s Storage module (backed by Amazon S3) allows the app to handle user-generated content like videos out-of-the-box[[4]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=4,anagement). Videos can be uploaded directly to S3, which is highly scalable and durable. Processing of those videos (transcoding, resizing, etc.) can be orchestrated via serverless functions (AWS Lambda triggered by S3 events) or AWS Elemental MediaConvert. Amplify can integrate such workflows using Lambda triggers and additional AWS services as needed (and even supports adding custom AWS resources if the default categories don’t cover a use-case[[5]](https://medium.com/@james.tosswill/navigating-ecs-deployments-via-aws-amplify-ci-cd-unravelling-the-why-what-and-how-fd82b0bccad7#:~:text=By%20integrating%20AWS%20Amplify%2C%20Lambda%2C,complexity%20in%20our%20deployment%20cycle)).
* **Data and APIs (Comments):** Amplify supports both GraphQL and REST API backends. Using **AWS AppSync (GraphQL)** or **API Gateway + Lambda (REST)**, Amplify can generate a backend API for comments. For instance, an Amplify project could define a GraphQL schema for a “Comment” model; Amplify will provision an AppSync API and DynamoDB table to store comments. This provides a highly scalable, serverless database (DynamoDB) that can handle large volumes of reads/writes with auto-scaling capacity. Alternatively, one can use Amplify’s “Functions” to create serverless REST endpoints (Node.js, Python, etc.) and connect to a database of choice (DynamoDB, Aurora Serverless, etc.)[[6]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=%E2%80%A2). These services scale automatically – e.g. DynamoDB on-demand mode scales throughput as needed, and Lambda functions scale by spawning concurrent executions under load.
* **Authentication & Security:** Amplify makes it straightforward to implement secure user authentication via **Amazon Cognito**. By running amplify add auth, developers get a fully managed user sign-up/sign-in flow with support for email/phone verification and social logins. Cognito enforces secure token-based access control, and Amplify’s client libraries integrate with it, simplifying authentication on the mobile app. This addresses the secure auth requirement with minimal custom work – **“For modern applications with minimal DevOps overhead, AWS Amplify makes sense,”** especially given its built-in Auth and integration with front-end frameworks[[7]](https://dev.to/techwithhari/aws-amplify-vs-azure-app-service-a-developers-perspective-3f2o#:~:text=For%20modern%20JavaScript%20applications%20with,overhead%2C%20AWS%20Amplify%20makes%20sense). Access control to the videos and comments can also leverage Cognito user identities (for example, S3 buckets can be configured to allow only the video owner or authorized users to access certain content via Cognito identity federation).
* **Scalability:** Amplify’s backend resources are serverless and managed by AWS. This means the app can handle growth without a complete re-architecture. Static content (if any web hosting) is served from S3/CloudFront which can handle massive traffic, and dynamic requests go to Lambda/AppSync which auto-scales. *Amplify itself does not impose scaling limits; it “uses managed services” (S3, Lambda, API Gateway/AppSync, etc.) for hosting and backend, so those services’ scaling characteristics apply*[*[8]*](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=%E2%80%A2%20%E2%80%A2%20Edited)*.* For instance, Lambda can scale from zero to thousands of concurrent executions nearly instantly for sudden traffic spikes[[9]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=AWS%20Lambda%20automatically%20spins%20up,unpredictable%20traffic%20spikes%20or%20requests), and AppSync/DynamoDB can handle very high request rates (with configurable limits)[[10]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=Many%20have%20addressed%20provided%20some,request%20to%20increase%20these%20limits). This serverless scalability is a strong advantage for unpredictable or spiky workloads – one AWS engineer notes that Amplify’s Lambda-based approach can work for everything from startups to large enterprise loads[[11]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=Hello%21%20It%27s%20Serverless%20all%20the,can%20help%20you%20design%20an). There is no need to pre-provision servers for peak capacity; the architecture naturally scales out as usage grows.
* **Developer Experience & Team Collaboration:** Amplify is designed to be developer-friendly and quick to iterate. Developers can use the Amplify CLI to add features (auth, API, storage, etc.) with guided prompts, and Amplify will provision the necessary AWS resources via CloudFormation under the hood. Code for serverless functions or resolvers can be edited and pushed with a single command (amplify push). This rapid, unified workflow means front-end and back-end can be developed together seamlessly – e.g. *“editing frontend and backend simultaneously, with Lambda code updating remotely while still available on the frontend” feels nearly like local development*[*[12]*](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=,can%20easily%20delete%20if%20needed)*.* Amplify also supports a **multi-environment workflow**, so each developer on a remote team can create their own isolated backend environment (e.g. *Dev1 has an “alice-dev” backend, Dev2 has “bob-dev”, etc.*) and work without interfering with each other. These environments can be easily created or torn down as they are just stacks of managed services, satisfying the need for flexible collaboration. One user highlights the **blazing fast development cycle** Amplify enables: auto-deploying updates on save, making cloud stack changes feel as quick as running a local server[[13]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=The%20cool%20thing%20I%20like,any%20specific%20solution%20around%20that). In short, Amplify abstracts much of the complexity, letting a distributed team focus on product features instead of wiring up infrastructure – *it “abstracted away the complexities of infrastructure management, enabling [developers] to focus on ... features” and iterate faster*[*[3]*](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=developers%20required%20a%20platform%20that,faster%2C%20iterate%20on%20concepts%20more).
* **Maintenance & Operations:** Because Amplify leverages fully managed services, operational overhead is very low. There are no servers or OS to manage, no container cluster to tune – AWS handles uptime, scaling, and patches for services like Lambda, DynamoDB, and AppSync. The Amplify team (or AWS) manages updates to these services and the Amplify framework itself. This means fewer routine tasks for the developers (no manual scaling, no OS patch Tuesday, etc.). Monitoring and logging are integrated (e.g. CloudWatch for Lambda logs, AWS Pinpoint or CloudWatch for analytics). As the **KnowledgeAcademy** notes, Amplify was **“born out of the need to simplify [the] complex process”** of full-stack development and reduce maintenance burdens[[2]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=AWS%20Amplify%20is%20a%20solution,the%20fiercely%20competitive%20digital%20landscape). That said, using Amplify’s CLI does introduce a dependency on that tool, and in very large projects some developers have reported challenges (Amplify’s older CLI had bugs scaling to large infrastructures[[14]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=I%27ve%20been%20moving%20away%20a,CLI%20is%20riddled%20with%20bugs)). AWS is addressing this with Amplify’s newer “GraphQL Transform v2 / Amplify Geo v2” that is built on CDK, aiming to improve maintainability. But for a mid-sized app, Amplify’s managed approach typically means far less ops work compared to managing containers or VMs.

## AWS ECS Fargate Overview for this Use Case

AWS ECS Fargate is a **container orchestration service** where AWS manages the server infrastructure while you run your application in Docker containers. In this architecture, we would containerize our backend services and use Terraform to define and provision the AWS resources (ECS cluster, tasks, networking, etc.). This approach provides more fine-grained control over the runtime and architecture at the cost of additional complexity. Here’s how an ECS Fargate solution would address the app’s needs:

* **Microservices & Custom Backend Logic:** Using ECS Fargate, we can create separate containerized services for different backend functions. For example, one containerized service might handle the **Comments API** (e.g. a Node.js or Python REST API server that interacts with a database for comments), and another service might handle **Video Processing** tasks (e.g. a service running FFmpeg or calling libraries to transcode videos). These services run on Fargate, meaning we do not manage EC2 servers; AWS runs the containers on our behalf in a serverless fashion. However, unlike Amplify, we are responsible for designing the service interactions, choosing the databases, and writing more boilerplate (e.g. writing the entire API server code rather than using a generated AppSync API). We also define CPU/memory for each task and how they scale. *In essence, Fargate gives us container-level control — including OS, runtime, networking settings — without managing physical servers*[*[15]*](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Setup%20and%20Maintenance)[*[16]*](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=With%20Lambda%2C%20you%20can%20write%2C,closest%20thing%20to%20serverless%20computing)*.* This can be useful if we have specific runtime requirements (e.g. using a specific video processing library or a language not directly supported in Lambda).
* **Video Storage & Processing:** In an ECS architecture, we would likely still leverage **Amazon S3** to store the raw and processed video files (since storing large video blobs in container filesystems is not practical). The mobile app could upload videos to S3 (perhaps via a presigned URL provided by a backend service). Once a video is in S3, we could use an event trigger (S3 event to SQS or EventBridge) to notify a **Video Processor** service running on Fargate. For instance, a Fargate task could be launched to process each video. This approach is more manual than Amplify’s automatic Lambda trigger, but it’s feasible: we’d configure S3 events and possibly use AWS ECS RunTask API to start a Fargate task for each new video (this pattern is actually possible to integrate with Amplify too[[5]](https://medium.com/@james.tosswill/navigating-ecs-deployments-via-aws-amplify-ci-cd-unravelling-the-why-what-and-how-fd82b0bccad7#:~:text=By%20integrating%20AWS%20Amplify%2C%20Lambda%2C,complexity%20in%20our%20deployment%20cycle), but in our pure-ECS scenario we set it up via Terraform). The video processing container can run longer tasks and utilize more CPU/memory than a Lambda (which is limited to 15 minutes execution). If videos are large or need complex processing, the Fargate container can handle it without the hard time limits. One trade-off: if many videos come in at once, ECS will queue up tasks; scaling is not instant but can be automated.
* **Comment Storage & API:** We would choose a database for comments – for example, an AWS RDS (Aurora) database or DynamoDB. In Terraform we’d provision that and our Comments API container would connect to it. If using DynamoDB, we lose Amplify’s automatic AppSync integration but can call DynamoDB via AWS SDK from our container code. If using RDS, we’d manage connections pooling, etc. The **Comments service** on ECS could be behind an Application Load Balancer (ALB) to expose REST endpoints to the mobile app. Cognito could be integrated by using ALB’s Cognito authentication feature or by validating JWT tokens in the container code. All of this gives us flexibility (any database, any auth method, any backend framework), but it requires writing and maintaining the glue code.
* **Scalability:** With ECS Fargate, scaling is achieved by adjusting the number of running container tasks. We would set up ECS Service Auto Scaling policies (based on CPU utilization, request count on the ALB, SQS queue length, etc.) to scale out the tasks when load increases[[17]](https://bobcares.com/blog/amplify-aws-fargate/#:~:text=AWS%20Fargate%20can%20be%20described,down%20according%20to%20the%20demands)[[18]](https://bobcares.com/blog/amplify-aws-fargate/#:~:text=When%20we%20use%20AWS%20Amplify,and%20run%20the%20containerized%20application). For example, the comment API service might scale between 2 to 10 containers based on traffic. Fargate can respond to scaling needs reasonably quickly, though not as instantly as serverless functions. Notably, recent improvements allow ECS on Fargate to launch up to 500 tasks in about 2 minutes under load[[19]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Before%20April%202022%2C%20AWS%20Fargate,especially%20when%20requests%20spiked%20suddenly). However, Fargate **does not scale to zero** on its own – you will always have at least one container of each service running to accept traffic (unless you manually shut it off)[[20]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=AWS%20Lambda%20automatically%20spins%20up,unpredictable%20traffic%20spikes%20or%20requests). This means for *very low-traffic periods, you might have idle containers still running (incurring cost), but it also means no cold-start latency when traffic arrives* (the containers are already warm). For thousands of users, a well-configured Fargate service can certainly handle the load; the throughput will depend on how many containers and their size. One advantage is **consistent performance for sustained loads** – since containers run continuously, there’s no per-request spin-up. But for **spiky or unpredictable traffic**, Fargate’s slightly slower scale-up (and need to predict capacity or risk being a step behind demand) is a consideration; AWS Lambda (Amplify’s approach) can scale more rapidly and granularly per request[[9]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=AWS%20Lambda%20automatically%20spins%20up,unpredictable%20traffic%20spikes%20or%20requests)[[21]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Lambda%2C%20however%2C%20has%20no%20maximum,limit%20change%20request%20to%20AWS). Also, certain limits exist (Fargate tasks max out at a certain CPU/RAM per container, historically 4 vCPU/30 GB RAM[[22]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Fargate%2C%20on%20the%20other%20hand%2C,4%20vCPUs%20for%20all%20applications), though you can run many in parallel). In summary, ECS Fargate can achieve the needed scale but requires tuning auto-scaling policies and might provision extra capacity to be safe, whereas Amplify’s serverless stack scales automatically by AWS.
* **Security & Authentication:** In a Fargate architecture we typically integrate AWS services for auth as well, but the integration isn’t as automatic as Amplify. We can use Amazon Cognito User Pools for the mobile app’s authentication (which provides JWT tokens upon login). Our containerized API would need to validate these tokens on each request (using Cognito’s public keys) or, if using an ALB, we could offload auth to the ALB by enabling Cognito authentication on that listener. Terraform can provision these integrations. All data storage (S3, databases) and inter-service communication would be confined to a VPC for security. We’d manage IAM roles for tasks to ensure least privilege (e.g. the video processing task can read from the input S3 bucket and write to an output bucket, etc.). Essentially, Fargate gives us all the building blocks to implement security best practices, but **it’s on the team to configure them**. Amplify, by contrast, sets up many of these by default (e.g. a Cognito auth and S3 storage with the proper auth integration).
* **Developer Experience & Collaboration:** Using ECS Fargate with Terraform means embracing an Infrastructure-as-Code and DevOps-heavy workflow. The development experience involves multiple components: writing application code (for the services), Dockerizing it, managing an AWS ECR repository for images, and writing Terraform scripts for all AWS resources. For a remote team, collaboration happens through code repositories: e.g. developers contribute to the application code and to the Terraform configurations. This approach has the benefit of explicit version-controlled infrastructure – changes are reviewed and applied through Terraform, which can be very robust for team collaboration (no one is manually clicking in AWS; everything is reproducible). However, the **iteration cycle** is slower. A code change must be rebuilt into a container image and deployed via the CI/CD pipeline to ECS, which takes a few minutes at best. Infrastructure changes require running Terraform plan/apply, which also takes time and can be complex. There’s more **overhead to set up CI/CD**: one needs to create pipelines (perhaps using AWS CodePipeline or GitHub Actions) to automate image builds and Terraform deploys. Amplify’s approach, in contrast, has much of this wired up for you (Amplify Console can auto-deploy on git push[[23]](https://dev.to/techwithhari/aws-amplify-vs-azure-app-service-a-developers-perspective-3f2o#:~:text=AWS%20Amplify%3A%20Frontend), and amplify push automates CloudFormation deployments). With ECS/Terraform, the team likely will containerize the app and might use Terraform Cloud or a CI runner to apply changes. Teams with strong DevOps practices may prefer this control – for example, they can impose code reviews on infrastructure changes and use Terraform modules for consistency. Each developer could run portions of the system locally (Docker Compose to simulate services with local DB, etc.) for testing, which is a plus of containers. But setting up **separate full environments per developer** is non-trivial; one could parameterize Terraform to create isolated dev stacks (with unique resource names) as one reddit user suggests[[24]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=skill), but it’s not as simple as Amplify’s built-in env feature. In summary, **the ECS/Fargate path offers greater flexibility and is aligned with traditional software engineering workflows, but has a steeper learning curve and more steps in the development loop**. It might be preferred if the team needs that fine-grained control or already has container expertise. If the team is primarily application developers without dedicated DevOps, this route could slow them down initially.
* **Maintenance & Operations:** Operating an ECS Fargate stack still incurs less maintenance than managing EC2 servers, but more than a fully managed Amplify stack. With Fargate, we must **monitor container health, CPU/memory usage, and scale settings**, update container images (e.g. apply security patches to base images), and handle logging/alerting setup (CloudWatch Logs, etc., which Terraform can configure). There’s also the matter of Terraform state management (ensuring remote state is set up for team use) and periodic Terraform module upgrades. AWS does manage the underlying servers for Fargate (so no OS patching on our part at the EC2 level), which is a big plus – *Fargate “abstracts the EC2 instances from you… freeing you from the operational burden of managing a fleet of servers (at a small premium)”*[*[25]*](https://dev.to/raphael_jambalos/secret-costs-of-ecs-fargate-4j3b#:~:text=You%20don%27t%20have%20access%20to,a%20fleet%20of%20EC2%20instances). In other words, Fargate trades a bit more cost for simpler ops compared to running your own EC2 cluster. But **compared to Amplify’s serverless approach, there are still more components to manage**. For example, if something goes wrong with a container (memory leak, crash), the team has to diagnose it, whereas in Amplify a lot of that is handled by AWS (Lambda would just start a new instance on failure). Overall, the ops overhead with Fargate is moderate – you don’t manage hardware, but you do manage everything above the container layer. Amplify’s ops overhead is minimal – you mostly manage your code and AWS takes care of the rest. An AWS engineer summarized this trade-off well: moving from serverless (Lambda/Amplify) to container or EC2 infrastructure can sometimes seem cost-effective at scale, *“but in my experience the resulting overhead of managing that infrastructure tends to equalize the two”*[[26]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=enterprises,want%20to%20compare%20the%20two). This suggests that any potential cost savings with a DIY Fargate approach might be offset by the manpower and complexity of operations, something particularly relevant for a small remote team.

Having outlined each approach, we can now directly compare them on the key evaluation criteria.

## Comparison of Key Metrics

To clarify the differences, the following table summarizes AWS Amplify vs. AWS ECS Fargate on the core criteria:

| **Aspect** | **AWS Amplify (Serverless)** | **AWS ECS Fargate (Containers)** |
| --- | --- | --- |
| **Cost Efficiency** | *Very low baseline cost.* Primarily pay-per-use for services – e.g. no running servers when idle, so development or low-traffic periods incur minimal cost. Underlying resources (Lambda, DynamoDB, etc.) have free tiers or scale-to-zero behavior. At scale, costs rise with usage; **per-request costs** can be higher than an equivalent continuously running server[[27]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=percentage%20of%20time%20utilization). (AWS Lambda’s CPU-second pricing is higher than Fargate’s per CPU-hour cost when fully utilized, roughly making Lambda ~2× cost of Fargate for the same compute if constantly busy[[27]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=percentage%20of%20time%20utilization).) However, if utilization is sporadic, Amplify is cost-effective – you only pay for what you use, with no need to pay for idle capacity[[28]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=Billing%20starts%20when%20the%20image,code%20returns%20or%20otherwise%20terminates). One study noted Lambda becomes cost-advantageous until a function is running >40-50% of the time, beyond which a container might be cheaper[[29]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=Both%20the%20EC2%20instance%20and,from%20DynamoDB%2C%20and%20so%20on). In summary, Amplify is **cheaper during development and low-to-medium traffic**, and can scale without upfront cost, but extremely heavy constant workloads may lead to higher bills than an optimized container setup. |  |
| **Cost Efficiency** *(continued)* | *Cost predictability:* Amplify’s cost is composed of many small charges (API calls, Lambda GB-seconds, etc.), which can complicate forecasting but scales linearly with usage. There is no need to over-provision – you won’t pay for unused capacity. | *Notable baseline cost.* Requires at least one or a few containers always running to serve requests (cannot auto-scale to zero load by itself). This means even a dev/test environment or off-peak hours incur some cost (e.g. a small Fargate task ~$10–$20/month each). At scale, **bulk throughput can be cheaper**: a container running at high utilization can be more cost-efficient than an equivalent load on Lambda[[26]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=enterprises,want%20to%20compare%20the%20two). You can also use ECS with Savings Plans or Fargate Spot for discounts[[30]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=However%2C%20Fargate%20lets%20you%20use,up%20to%2072). But you must manage capacity – over-provisioning (running more CPU/RAM than needed) will waste money. In practice, Fargate is often **more economical for sustained high-load** scenarios, whereas Amplify is economical for bursty or low-load scenarios. Teams must also factor the **DevOps cost** – the effort to manage infrastructure (which doesn’t show on AWS bill but is real). As AWS’s own staff hinted, the operational overhead of containers can offset some raw cost savings[[26]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=enterprises,want%20to%20compare%20the%20two). |
| **Performance & Scalability** | *Highly scalable and burstable.* Built on AWS Lambda, API Gateway/AppSync, etc., Amplify can automatically scale to thousands of concurrent requests without manual intervention. **Scaling is elastic:** new instances of functions or increased DynamoDB throughput are allocated by AWS on demand. Cold starts are a consideration (initial requests to a Lambda may have a few hundred milliseconds delay), but after warm-up, performance is smooth. For unpredictable or spiky traffic, this model excels – AWS Lambda can scale from 0 to 1000+ executions almost instantly[[9]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=AWS%20Lambda%20automatically%20spins%20up,unpredictable%20traffic%20spikes%20or%20requests), and just as importantly scale **back to 0** when traffic subsides, saving cost. There is essentially no fixed limit on concurrent execution aside from account limits (which can be raised)[[31]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Lambda%2C%20however%2C%20has%20no%20maximum,limit%20change%20request%20to%20AWS). Amplify’s AppSync and storage services similarly handle high throughput (with configurable limits and auto-scaling on the data layer). A well-architected Amplify backend (using DynamoDB, S3, etc.) can achieve very high performance – one AWS engineer noted that *“100 req/s or basically anything you could imagine… is negligible [for serverless] as long as you’re able to pay for it”*[[32]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=At%20AWS%27s%20scale%2C%20100%20req%2Fs,able%20to%20pay%20for%20it). Video processing in Amplify’s serverless paradigm might require asynchronous pipelines (e.g. trigger a series of Lambda tasks or use AWS MediaConvert) due to the 15-minute Lambda cap, but those AWS services themselves scale in parallel. Overall, Amplify’s serverless model offers **rapid scaling and minimal tuning**, ideal for apps expecting variable load. | *Scalable with configuration.* ECS Fargate scales by running more container instances (tasks) for a service. It supports **steady high throughput** and long-running processes. There is no cold start per request – containers handle many requests, so latency can be lower and more consistent for sustained traffic. However, **scale-out is slower:** launching new containers can take tens of seconds to a couple of minutes, especially if pulling large images[[19]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Before%20April%202022%2C%20AWS%20Fargate,especially%20when%20requests%20spiked%20suddenly). AWS has improved this (up to 500 tasks in 2 minutes in newer updates[[33]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Before%20April%202022%2C%20AWS%20Fargate,especially%20when%20requests%20spiked%20suddenly)), but it’s not instantaneous like Lambda. Thus, for sudden spikes, Fargate may need headroom or predictive scaling. Fargate does **not auto-scale to zero** load by itself – you’d typically keep 1 task running to handle any traffic[[20]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=AWS%20Lambda%20automatically%20spins%20up,unpredictable%20traffic%20spikes%20or%20requests). This ensures zero cold-start latency at the expense of running idle capacity. Fargate has some resource limits per task (vCPU, memory, no GPU support yet), but you can run many tasks in parallel. It shines for **workloads that need continuous processing or specialized libraries** running for extended periods (e.g. processing a large video file for 30 minutes, which Lambda cannot do). In summary, ECS Fargate can achieve the needed scale for thousands of users, but it requires tuning auto-scaling policies and may have a bit more latency to ramp up. Once scaled out, performance is stable and it can handle sustained loads reliably (each container can be optimized for throughput). |
| **Developer Experience** <br>*(Remote Team)* | *Developer-friendly, fast iteration.* Amplify is designed for high productivity, especially for front-end or full-stack developers. The learning curve is relatively low – you can add complex capabilities (auth, storage, APIs) with CLI commands and minimal boilerplate. **Rapid prototyping** is a forte: *“Amplify allows developers to bring ideas to life faster and iterate more rapidly by abstracting infrastructure”*[[3]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=developers%20required%20a%20platform%20that,faster%2C%20iterate%20on%20concepts%20more). For a remote team, Amplify offers an out-of-the-box workflow to create multiple backend environments (dev, staging, prod) and even per-developer environments easily. Team members can collaborate by using Amplify CLI to pull and push environment updates; underlying configuration is stored as code (CloudFormation templates in the project) which can be version-controlled. The **tight integration between front-end and back-end** (with Amplify libraries) means front-end developers can implement backend features (like a new API endpoint or database model) with only a few CLI steps and some business logic code, without needing deep AWS expertise. The **feedback loop** is very quick: for example, updating a Lambda function’s code and seeing it live can be done in one command, and Amplify supports features like offline development and mocking for certain services. One developer noted that development with Amplify felt *“blazing fast…almost like [services] running locally”* due to auto-update features[[13]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=The%20cool%20thing%20I%20like,any%20specific%20solution%20around%20that). This speed is a big advantage for iterative development and testing. *Collaboration-wise*, because Amplify handles the heavy lifting, even a small or less DevOps-savvy team can manage a cloud backend. It reduces the need for a dedicated infrastructure engineer early on. On the flip side, Amplify’s opinionated structure means less flexibility; as projects grow, some developers have found the tool “clunky” or limiting for edge cases[[14]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=I%27ve%20been%20moving%20away%20a,CLI%20is%20riddled%20with%20bugs). But for the given use case, Amplify’s developer experience will likely be smoother and require **less setup and troubleshooting**, enabling the remote team to focus on app features rather than infrastructure details. | *Flexible but DevOps-heavy.* An ECS Fargate + Terraform approach gives the development team full control over the tech stack, which can be positive for experienced teams but presents a higher complexity. Developers (or DevOps engineers) must be comfortable with Docker, AWS networking (VPC, subnets, security groups), and Terraform syntax. Setting up the project initially is more involved: you will create Terraform scripts for the ECS cluster, task definitions, load balancer, database, etc., and Dockerfiles for each service. The **iteration cycle** is slower – every code change requires container build/push and an update deployment. With proper CI/CD (which the team must set up, e.g. using CodePipeline, Jenkins, or GitHub Actions), deployments can be automated on git push, but expect a few minutes for builds and rolling updates. In contrast to Amplify’s one-click environment creation, creating isolated stacks for each developer or feature branch with Terraform, while possible, demands careful configuration and resource naming to avoid collisions[[24]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=skill). For a remote team, coordination through code reviews and merges is critical to avoid breaking shared resources. On the plus side, everything is expressed as code (both infrastructure and application), which improves transparency. Developers can run containers locally to test (which is closer to production behavior than Amplify’s local mocks). They can also use any programming language or framework inside containers (Amplify is more tuned to JavaScript/TypeScript and a set of supported tech). **In summary**, the ECS/Terraform path provides a **“enterprise-like” developer experience – greater flexibility, more moving parts – which can be beneficial if your team has the skillset and needs that flexibility (custom runtime, specific frameworks, etc.).** However, for a small remote team aiming for speed and simplicity, this approach can slow down feature development due to the overhead of managing and understanding the infrastructure. |
| **Maintenance & Operations** | *Minimal ops burden.* Since Amplify uses fully managed services (Lambda, DynamoDB, AppSync, etc.), there are very few infrastructure maintenance tasks. AWS handles server provisioning, scaling, patching, and failover automatically[[34]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Lambda%20hosts%20and%20runs%20your,vCPU%20capacity%2C%20and%20monitoring%20logs). The team’s responsibilities are largely limited to application logic, configuring scaling rules (if needed) at a high level (e.g. setting DynamoDB auto-scaling), and monitoring the application metrics. Routine tasks like applying OS security updates or resizing clusters are eliminated. This makes Amplify ideal for a small team without a dedicated ops engineer – one can essentially **“set it and forget it”** for many infrastructure concerns. The trade-off is reliance on AWS’s managed defaults; you trust AWS to handle uptime and scaling (which they generally do well). If something goes wrong in a managed service, you have less direct control to fix it (you’d work with AWS support). Amplify also packages deployments via CloudFormation, which can be slow for very large stacks and occasionally tricky to debug if deployments fail. But these issues are infrequent in moderate-sized projects. The **bottom line** is that Amplify was explicitly created to **reduce maintenance overhead and complexity** in the development process[[2]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=AWS%20Amplify%20is%20a%20solution,the%20fiercely%20competitive%20digital%20landscape). One developer’s experience over 3 years confirms it “enabled delivering features and managing operations effectively” as a lean team[[35]](https://medium.com/@BryMei/after-3-years-of-aws-amplify-is-it-the-best-backend-as-a-service-4f12bed21511#:~:text=As%20a%20one,I%E2%80%99ve%20encountered%20along%20the%20way). In operations, Amplify provides out-of-the-box monitoring via CloudWatch and an admin UI for certain tasks (like managing users in Cognito), further easing operational efforts. | *Moderate ops overhead.* With ECS Fargate, there are fewer ops tasks than running your own servers, but you still have to **manage the container ecosystem**. This includes: monitoring container health and performance, handling deployments (like dealing with failed deployments or container crashes), and updating base images for security. Terraform configuration needs upkeep (ensuring modules and AWS provider are up-to-date, managing the Terraform state file safely). While AWS handles the underlying machine fleet for Fargate, the team must handle everything at the container level and above. For instance, logs from containers need to be routed to CloudWatch or another system – an extra config step. Scaling policies must be tested and tuned to balance cost and performance. There’s also the need for ongoing cost monitoring; e.g. a misconfigured service can run up bills (as seen when a misconfig caused continuous container restarts and massive bandwidth charges)[[36]](https://dev.to/raphael_jambalos/secret-costs-of-ecs-fargate-4j3b#:~:text=This%20is%20where%20we%20got,charged%20so%20much)[[37]](https://dev.to/raphael_jambalos/secret-costs-of-ecs-fargate-4j3b#:~:text=One%20of%20my%20past%20times,went%20from%2038GB%2Fmo%20to%2016TB%2Fmo). On the positive side, many operational tasks can be automated or handled with IaC – once the Terraform and CI pipelines are in place, managing changes is systematic. AWS Fargate frees you from managing EC2 instances (no capacity planning at the VM level, no cluster patching), which is a significant reduction in ops compared to ECS on EC2. *“For a bit of a premium, you are freed from the operational burden of managing a fleet of EC2 instances”* with Fargate[[25]](https://dev.to/raphael_jambalos/secret-costs-of-ecs-fargate-4j3b#:~:text=You%20don%27t%20have%20access%20to,a%20fleet%20of%20EC2%20instances). So you trade some higher cost for less ops work, which often is worth it. Compared to Amplify, however, the ops overhead remains higher: your team will spend time on infrastructure code, container management, and dealing with any AWS infrastructure issues yourselves. This approach might make sense if you have stringent requirements that Amplify can’t meet or if you plan to invest in DevOps for long-term control. But if minimizing ops is the goal, Amplify has the clear edge. |
| **Deployment Speed & CI/CD** | *Fast deployments, integrated pipeline.* Amplify offers very quick setup for continuous deployment. The **AWS Amplify Console** can connect to your code repository and automatically build and deploy the application backend and frontend on each commit[[23]](https://dev.to/techwithhari/aws-amplify-vs-azure-app-service-a-developers-perspective-3f2o#:~:text=AWS%20Amplify%3A%20Frontend). This means the team can have a CI/CD pipeline with almost zero effort – every git push triggers a new deployment (with build & test phases configurable). Even without Amplify Console, using the CLI to push updates is straightforward and fast for small changes (e.g. updating a Lambda function or adjusting an AppSync schema). Because there are no custom servers to roll out, deployments often involve updating configurations or code in managed services (which happens in seconds to a few minutes via CloudFormation changesets). The speed of deploying to Lambda or AppSync is generally faster than deploying to ECS because there’s no container image build unless you explicitly use container images for Lambdas. During development, developers can also leverage Amplify’s **Hot Reload-like** capabilities (for front-end hosting) and manual amplify mock or amplify push for backends to see changes quickly. This fosters an **agile, iterative release process**. You can deploy small updates multiple times a day without much hassle, and spinning up new environments (for testing a feature) is automated. Overall, Amplify accelerates the “code to cloud” cycle, which is valuable for a remote team working asynchronously – changes can be deployed and tested in real time, keeping everyone aligned. | *Slower deployments, custom pipeline needed.* With ECS/Fargate, setting up CI/CD is part of the project. The team would need to configure a pipeline that: builds Docker images for each service, pushes them to Amazon ECR, applies Terraform (or updates ECS service definitions) to deploy the new images, and possibly runs database migrations. Each of these steps adds time. A typical container build and ECS deploy might take several minutes (especially if images are large). Terraform apply can also take time if it needs to modify AWS resources. So, whereas Amplify might deploy a code change in, say, 30 seconds to a minute, the ECS path might take 5–10 minutes or more for a full pipeline run. During this time, developers have to wait to see their changes live. For frequent iterations, this lag can slow down feedback loops. There are services like AWS CodeDeploy/CodePipeline or third-party CI tools to automate this, but they need to be implemented and maintained. The upside is that once such a pipeline is in place, deployments are **consistent and under your control** – you can implement blue/green deploys, canary releases, etc., which are more manual to achieve with Amplify’s simpler deployment model. However, those advanced strategies may be overkill for this use case. In terms of initial **time-to-market**, Amplify likely allows much faster deployment of the first version (since so many pieces are pre-configured), whereas a from-scratch ECS setup will require a significant upfront effort to get all moving parts working (Terraform scripts, security configurations, CI pipeline). For a team aiming to **rapidly release** a new version of an app, Amplify’s speed of deployment is a major advantage. |

*(Table legend: AWS Amplify refers to using Amplify’s framework which leverages services like AWS Lambda, AppSync, S3, Cognito, etc. AWS ECS Fargate refers to a solution with containerized services orchestrated by ECS on Fargate, with Terraform managing infrastructure as code.)*

## Recommendation and Conclusion

Considering the requirements and the comparison above, **AWS Amplify is generally the better-suited architecture for this mobile app’s backend (Version 2)** given the context of a remote development team, need for fast iteration, and desire for low operational overhead. Amplify’s serverless, managed approach aligns well with the project’s needs in several ways:

* **Rapid Development & Feature Delivery:** Amplify will allow the team to add features like auth, file storage, and APIs in a matter of days (or hours) with minimal boilerplate. This speed is crucial for an agile rollout of V2. The development experience is simplified, which is beneficial for a remote team that might not have a dedicated DevOps engineer. In contrast, an ECS Fargate solution, while powerful, would require a longer setup phase and ongoing devops work for each change, potentially slowing down feature development and iteration speed[[7]](https://dev.to/techwithhari/aws-amplify-vs-azure-app-service-a-developers-perspective-3f2o#:~:text=For%20modern%20JavaScript%20applications%20with,overhead%2C%20AWS%20Amplify%20makes%20sense)[[13]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=The%20cool%20thing%20I%20like,any%20specific%20solution%20around%20that).
* **Low Maintenance & Team Overhead:** With Amplify, much of the heavy lifting (scaling, patching, infrastructure management) is offloaded to AWS’s managed services[[34]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Lambda%20hosts%20and%20runs%20your,vCPU%20capacity%2C%20and%20monitoring%20logs)[[8]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=%E2%80%A2%20%E2%80%A2%20Edited). This means the team can remain small and focused on application logic. The remote developers can each work on their own instance of the backend without worrying about complex deployments. The operational simplicity is a big win, as it reduces the cognitive load and chances of configuration errors. By contrast, running containers on Fargate introduces more points of maintenance (CI pipelines, container health, Terraform state), which could be burdensome for a team that values low ops overhead.
* **Scalability and Performance for the Use Case:** Amplify’s architecture will comfortably handle thousands of users and can leverage AWS services optimized for high scale (S3 for videos, DynamoDB for comments, etc.). Unless the application has very specific performance needs (e.g. requiring GPU for video processing or extremely long-running processes), Amplify’s serverless model should perform well. It’s worth noting that many successful large-scale apps use similar serverless patterns. Fargate would also meet the scalability requirements, but it doesn’t provide a clear performance advantage for the described workload to justify the extra complexity. In fact, for unpredictable spikes (e.g. a viral video causing a surge in uploads), Amplify/Lambda would automatically scale out more rapidly[[9]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=AWS%20Lambda%20automatically%20spins%20up,unpredictable%20traffic%20spikes%20or%20requests) than an ECS cluster might.
* **Cost Considerations:** During development and testing, Amplify will be very cost-efficient – you pay almost nothing when the app is not heavily used, which is ideal while building and refining V2. There is no need to run always-on servers for dev or low-traffic periods. As usage grows, costs will increase, but they will directly correspond to actual user activity (e.g. storage costs for the volume of videos, Lambda/AppSync costs per request, etc.). Fargate, on the other hand, has a relatively higher baseline cost (you’d be paying for containers to be up even if only a few users are active at a given moment). If the application eventually reaches a point where it has sustained high load 24/7 and the bills for serverless become a concern, the team could reconsider architecture then. As one commenter suggested, Amplify (serverless) is great for getting to market and handling growth, and only once *“you start getting so much traffic that costs become a problem”* would you look into a container or EC2 solution[[38]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=%E2%80%A2). It’s often wise to cross that bridge when you come to it, rather than prematurely optimizing for cost at the expense of development velocity.
* **Team Skill and Focus:** Finally, the recommendation considers the human factor. If the team’s expertise is more in application development (and perhaps they used Amplify or a similar BaaS for version 1), sticking with Amplify lets them leverage that familiarity. It also minimizes context-switching between writing application code and dealing with infrastructure intricacies. Terraform and ECS are powerful tools, but they might divert attention into infrastructure issues that don’t directly deliver user value. Amplify provides a lot of “pre-made” infrastructure as a service. Given that one goal is remote collaboration with minimal friction, Amplify’s guided approach reduces the chance of environment drift or deployment mistakes, which is very helpful when team members are not co-located.

**When Might ECS Fargate be chosen?** The ECS Fargate + Terraform approach could be justified if the project had requirements that Amplify cannot easily fulfill – for example, if the video processing required specialized native libraries or longer processing times that outstrip Lambda’s capabilities, or if the team needed a custom tech stack not supported by Amplify (say, a specific language runtime or a need for fine-grained network control). It also offers more flexibility for implementing sophisticated deployment strategies or integrating third-party services (though Amplify can integrate custom resources too[[5]](https://medium.com/@james.tosswill/navigating-ecs-deployments-via-aws-amplify-ci-cd-unravelling-the-why-what-and-how-fd82b0bccad7#:~:text=By%20integrating%20AWS%20Amplify%2C%20Lambda%2C,complexity%20in%20our%20deployment%20cycle)). Additionally, if the target was to optimize costs at very high scale with reserved capacity, a container approach could be tuned to be cheaper (assuming the team has the resources to manage it). But these factors don’t appear to outweigh Amplify’s advantages for the described scenario.

**Conclusion:** For a mobile app backend handling user videos and comments, with a need for quick development, easy scaling, and low ops burden, **AWS Amplify is the recommended choice**. It provides a cohesive, managed environment that covers authentication, storage, and APIs with minimal setup, and it will scale transparently to thousands of users[[11]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=Hello%21%20It%27s%20Serverless%20all%20the,can%20help%20you%20design%20an)[[8]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=%E2%80%A2%20%E2%80%A2%20Edited). This allows the remote development team to ship features faster and collaborate smoothly, without getting bogged down in infrastructure management. AWS ECS Fargate is a robust alternative that offers more control and might better suit a later stage of the product or a larger engineering organization, but for now, Amplify’s serverless architecture is better aligned with the project’s priorities and constraints.

**Sources:** The comparison above is based on information from AWS product documentation and pricing, as well as real-world experiences and analyses (cited inline). Key references include discussions by AWS engineers about Amplify’s scalability[[11]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=Hello%21%20It%27s%20Serverless%20all%20the,can%20help%20you%20design%20an)[[8]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=%E2%80%A2%20%E2%80%A2%20Edited), cost analysis articles contrasting Lambda (used by Amplify) vs. Fargate costs[[27]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=percentage%20of%20time%20utilization)[[29]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=Both%20the%20EC2%20instance%20and,from%20DynamoDB%2C%20and%20so%20on), and developer reports on Amplify’s and Terraform’s impact on team productivity[[3]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=developers%20required%20a%20platform%20that,faster%2C%20iterate%20on%20concepts%20more)[[13]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=The%20cool%20thing%20I%20like,any%20specific%20solution%20around%20that). These sources collectively reinforce the assessment that Amplify delivers on the requirements of fast, scalable, and low-maintenance development, whereas ECS Fargate caters to scenarios demanding greater control or heavy continuous workloads, which seem secondary in this context.

[[1]](https://bobcares.com/blog/amplify-aws-fargate/#:~:text=On%20the%20other%20hand%2C%20AWS,for%20building%20and%20deploying%20applications) [[17]](https://bobcares.com/blog/amplify-aws-fargate/#:~:text=AWS%20Fargate%20can%20be%20described,down%20according%20to%20the%20demands) [[18]](https://bobcares.com/blog/amplify-aws-fargate/#:~:text=When%20we%20use%20AWS%20Amplify,and%20run%20the%20containerized%20application) AWS Amplify AWS Fargate | The Complete Solution

<https://bobcares.com/blog/amplify-aws-fargate/>

[[2]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=AWS%20Amplify%20is%20a%20solution,the%20fiercely%20competitive%20digital%20landscape) [[3]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=developers%20required%20a%20platform%20that,faster%2C%20iterate%20on%20concepts%20more) [[4]](https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/#:~:text=4,anagement) What is AWS Amplify? Its Benefits and Features

<https://www.theknowledgeacademy.com/blog/what-is-aws-amplify/>

[[5]](https://medium.com/@james.tosswill/navigating-ecs-deployments-via-aws-amplify-ci-cd-unravelling-the-why-what-and-how-fd82b0bccad7#:~:text=By%20integrating%20AWS%20Amplify%2C%20Lambda%2C,complexity%20in%20our%20deployment%20cycle) Navigating ECS Deployments via AWS Amplify CI/CD: Unravelling the Why, What, and How | by James | Medium

<https://medium.com/@james.tosswill/navigating-ecs-deployments-via-aws-amplify-ci-cd-unravelling-the-why-what-and-how-fd82b0bccad7>

[[6]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=%E2%80%A2) [[12]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=,can%20easily%20delete%20if%20needed) [[13]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=The%20cool%20thing%20I%20like,any%20specific%20solution%20around%20that) [[14]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=I%27ve%20been%20moving%20away%20a,CLI%20is%20riddled%20with%20bugs) [[24]](https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/#:~:text=skill) Developers Who've Moved On From AWS Amplify - What Did You Switch To? : r/aws

<https://www.reddit.com/r/aws/comments/1e1balm/developers_whove_moved_on_from_aws_amplify_what/>

[[7]](https://dev.to/techwithhari/aws-amplify-vs-azure-app-service-a-developers-perspective-3f2o#:~:text=For%20modern%20JavaScript%20applications%20with,overhead%2C%20AWS%20Amplify%20makes%20sense) [[23]](https://dev.to/techwithhari/aws-amplify-vs-azure-app-service-a-developers-perspective-3f2o#:~:text=AWS%20Amplify%3A%20Frontend) AWS Amplify vs Azure App Service: A Developer's Perspective - DEV Community

<https://dev.to/techwithhari/aws-amplify-vs-azure-app-service-a-developers-perspective-3f2o>

[[8]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=%E2%80%A2%20%E2%80%A2%20Edited) [[10]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=Many%20have%20addressed%20provided%20some,request%20to%20increase%20these%20limits) [[11]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=Hello%21%20It%27s%20Serverless%20all%20the,can%20help%20you%20design%20an) [[26]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=enterprises,want%20to%20compare%20the%20two) [[32]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=At%20AWS%27s%20scale%2C%20100%20req%2Fs,able%20to%20pay%20for%20it) [[38]](https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/#:~:text=%E2%80%A2) How does AWS Amplify handle scaling of your app and servers, to handle growing traffic, unexpected load : r/aws

<https://www.reddit.com/r/aws/comments/lo8ur4/how_does_aws_amplify_handle_scaling_of_your_app/>

[[9]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=AWS%20Lambda%20automatically%20spins%20up,unpredictable%20traffic%20spikes%20or%20requests) [[15]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Setup%20and%20Maintenance) [[16]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=With%20Lambda%2C%20you%20can%20write%2C,closest%20thing%20to%20serverless%20computing) [[19]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Before%20April%202022%2C%20AWS%20Fargate,especially%20when%20requests%20spiked%20suddenly) [[20]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=AWS%20Lambda%20automatically%20spins%20up,unpredictable%20traffic%20spikes%20or%20requests) [[21]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Lambda%2C%20however%2C%20has%20no%20maximum,limit%20change%20request%20to%20AWS) [[22]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Fargate%2C%20on%20the%20other%20hand%2C,4%20vCPUs%20for%20all%20applications) [[30]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=However%2C%20Fargate%20lets%20you%20use,up%20to%2072) [[31]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Lambda%2C%20however%2C%20has%20no%20maximum,limit%20change%20request%20to%20AWS) [[33]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Before%20April%202022%2C%20AWS%20Fargate,especially%20when%20requests%20spiked%20suddenly) [[34]](https://www.cloudzero.com/blog/fargate-vs-lambda/#:~:text=Lambda%20hosts%20and%20runs%20your,vCPU%20capacity%2C%20and%20monitoring%20logs) Fargate Vs. Lambda: Battle Of The Serverless

<https://www.cloudzero.com/blog/fargate-vs-lambda/>

[[25]](https://dev.to/raphael_jambalos/secret-costs-of-ecs-fargate-4j3b#:~:text=You%20don%27t%20have%20access%20to,a%20fleet%20of%20EC2%20instances) [[36]](https://dev.to/raphael_jambalos/secret-costs-of-ecs-fargate-4j3b#:~:text=This%20is%20where%20we%20got,charged%20so%20much) [[37]](https://dev.to/raphael_jambalos/secret-costs-of-ecs-fargate-4j3b#:~:text=One%20of%20my%20past%20times,went%20from%2038GB%2Fmo%20to%2016TB%2Fmo) How we lost $800/mo with Amazon ECS Fargate - DEV Community

<https://dev.to/raphael_jambalos/secret-costs-of-ecs-fargate-4j3b>

[[27]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=percentage%20of%20time%20utilization) [[28]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=Billing%20starts%20when%20the%20image,code%20returns%20or%20otherwise%20terminates) [[29]](https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb#:~:text=Both%20the%20EC2%20instance%20and,from%20DynamoDB%2C%20and%20so%20on) Compare The Cost of AWS Lambda, Fargate, and EC2 For Your Workloads | by Milan Gatyás | Life at Apollo Division | Medium

<https://medium.com/life-at-apollo-division/compare-the-cost-of-aws-lambda-fargate-and-ec2-for-your-workloads-ad112c4740fb>

[[35]](https://medium.com/@BryMei/after-3-years-of-aws-amplify-is-it-the-best-backend-as-a-service-4f12bed21511#:~:text=As%20a%20one,I%E2%80%99ve%20encountered%20along%20the%20way) After 3 years of AWS Amplify, is it the Best Backend as a Service? | by Bryson Meiling | May, 2025 | Medium

<https://medium.com/@BryMei/after-3-years-of-aws-amplify-is-it-the-best-backend-as-a-service-4f12bed21511>