



UNUSUAL  
VENTURES



TECH SPOTLIGHT

# Serverless Technology

Technology is always evolving. Some advances make us turn our heads, but others rise above and fundamentally change the way we live and work. In this series we highlight an emerging technology that we believe has an unrivaled opportunity for startups.

BY JORDAN SEGALL & ABHI SHARMA



# Table of Contents

<b>INTRODUCTION</b>	<b>3</b>	<b>CURRENT MARKET LANDSCAPE</b>	<b>23</b>
Background	4	Platforms: Refinery.io	27
What is serverless technology?	4	Monitoring, logging, and debugging: Dashbird	27
Adoption rates	4	Deployment: Stackery	27
The serverless community	5	Security: Protego	27
<b>HOW SERVERLESS WORKS</b>	<b>6</b>	<b>CLOSING THOUGHTS</b>	<b>28</b>
Origins	6	<b>CONTACT</b>	<b>29</b>
Fundamental pillars	7		
Use case: Photo sharing	8		
<b>BENEFITS OF SERVERLESS</b>	<b>12</b>		
Big cost savings	12		
Improved developer experience	13		
Reduced overhead	13		
New marketplaces	13		
<b>BARRIERS TO ADOPTION</b>	<b>14</b>		
Performance	15		
Expressibility	18		
Operational monitoring and testing	19		
Deployment	20		
Vendor lock-in	20		
Security and privacy	21		
Runtime limitations	21		
Confusing costs	22		
Duration	22		
Memory	22		
Additional costs	22		

# Introduction

At Unusual Ventures, we believe serverless computing is the next wave of cloud computing and the next iteration of microservice-based architectures. We are excited to see what technologies emerge to better enable developers to leverage serverless within their organizations.

Just like the rise of cloud computing that allowed hundreds of successful startups to enhance functionality, we are seeing startups today built on top of serverless platforms like AWS Lambda Functions, Microsoft Azure Functions, and Google Cloud Functions.

We believe the serverless movement is just getting started and that it's going to be big.

## Overview

Define serverless technology and its evolution

The barriers preventing it from mass adoption today

Why it is an exciting (yet relatively early) emerging space

Examples of serverless companies that are paving the way for future adoption

Why serverless may not be the best fit for a given application

# Background

## What is serverless technology?

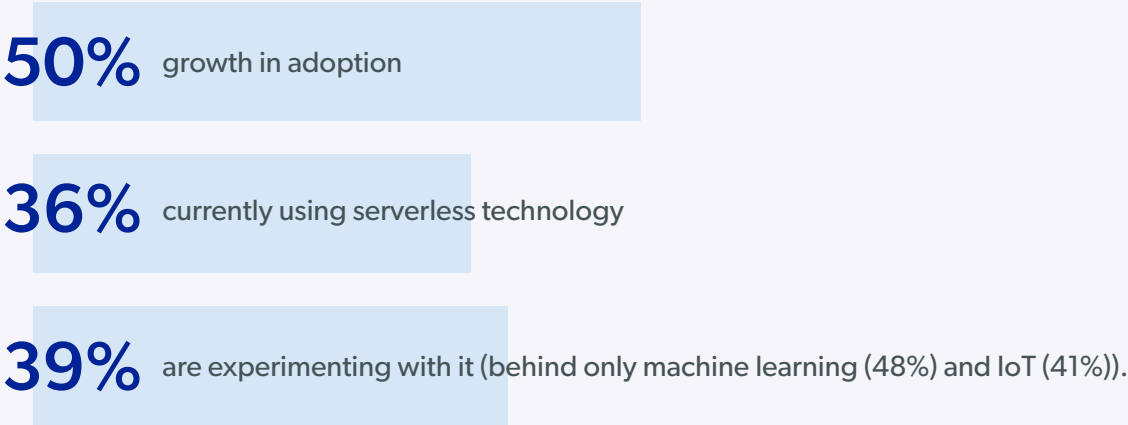
Serverless technology is not the absence of servers, but the absence of having to worry or even think about them. It transfers the management of servers and underlying operational infrastructure away from busy developers and over to established cloud / service providers.

This means developers no longer have to worry about time-consuming software deployment concerns like provisioning, utilization, scalability, fault-tolerance, and monitoring. Serverless raises the level of programming abstraction to a point that the data center becomes the computer. This is an incredibly powerful idea—one that comes with exciting opportunities and systems challenges.

## Adoption rates

Many companies have started this adoption. For the past two years, serverless technology has emerged as the fastest-growing service among public cloud services<sup>1</sup>.

## 2019 State of Serverless Technology



<sup>1</sup> Source: RightScale's 2019 State of the Cloud Report

# The serverless community

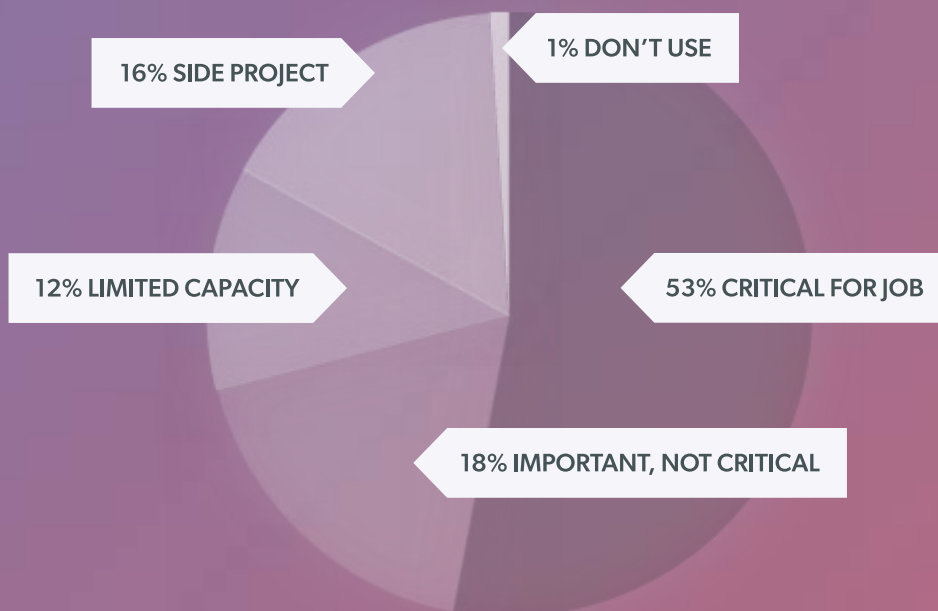
Companies across industries like Netflix, AutoDesk, Nordstrom, and Thomson Reuters have already adopted serverless as major parts of their architecture. In turn, they have experienced both significant cost reductions and ease of development and maintenance.

## Usage of serverless technology over time<sup>2</sup>

**45%** of respondents said they were using serverless at work in some capacity in 2017

**82%** of respondents said they were using serverless at work in 2018

## Importance of serverless at work

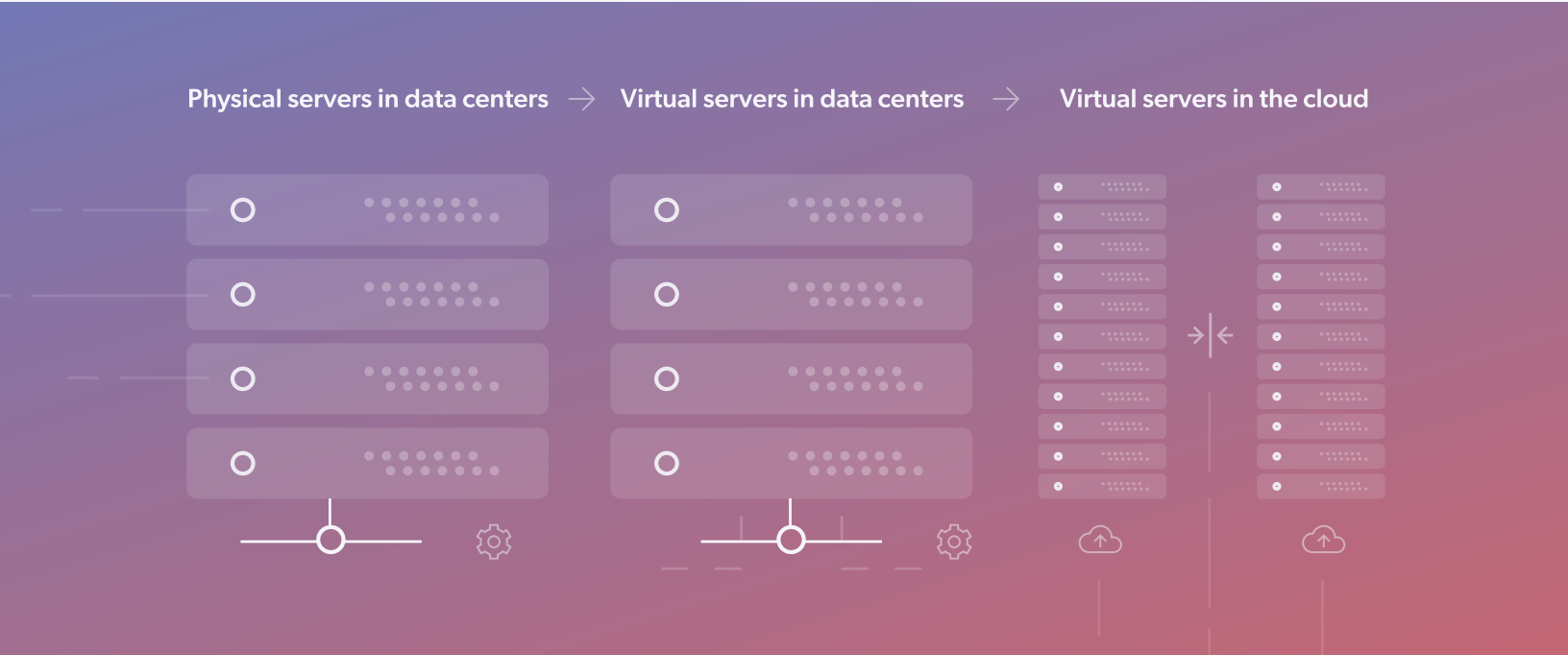


<sup>2</sup> Source: 2018 Serverless Community Survey: Huge Growth in Serverless Usage

# How serverless works

## Origins

The evolution of computing from a server infrastructure standpoint can be summarized at a high level as transitioning from physical servers in data centers, to virtual servers in data centers, to virtual servers in the cloud.



Each progressive step offers numerous advantages, such as higher utilization, faster provisioning, elastic resourcing, reduced infrastructure maintenance, hardware independence, increased availability, and trading CAPEX to OPEX.

These advantages were at the heart of the cloud computing revolution, but significant challenges remain. Notably, practitioners must still select and administer instances, manage capacity and utilization, control scalability, enable fault tolerance, and monitor systems. Serverless, the next step in this evolution, aims at alleviating these responsibilities completely. With serverless, services provision, scale, and monitor automatically.

3 **Source:** "Serverless Computing: Redefining the Cloud", Roger Barga, Serverless Conf 2017





# Fundamental pillars

FaaS (Functions-as-a-Service) or BaaS (Backend-as-a-Service) are fundamental pillars of serverless computing. You can think of FaaS as the computer model and BaaS as the storage model that enables serverless computing.

In this worldview, a traditional monolithic codebase is broken up into individual functions that are executed independently in the cloud. In essence, serverless/FaaS hinges upon an “event-driven” application structure that spins up servers on demand. As a result, resources are only allocated when these events trigger “functions”.

Examples of serverless events/triggers could be arrival of data, passage of time, upstream call, cron job, state change in a workflow, file upload to blob stores, or streaming events. Triggers can also occur via incoming HTTP API requests, which are handled by an API Gateway and routed to an appropriate function.

Since servers are de-allocated when function execution ends, any state that is created inside a function has to be externalized outside of the FaaS instance to be persisted for future usage. This externalized persistence is handled by BaaS, the other pillar of serverless, which offers persistence abstractions as an API. Popular serverless providers are Amazon, Google, and Microsoft.

# Use case: Photo sharing

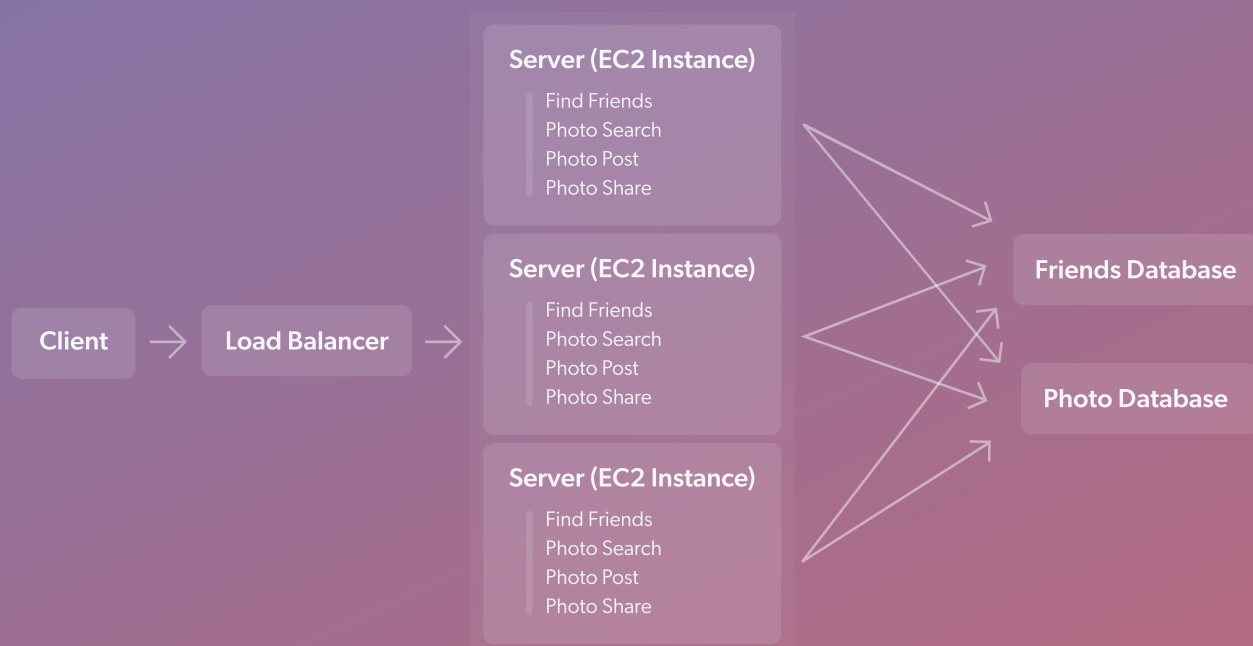
## Underlying process

As a simple example, imagine you want to enable a machine learning classifier to tell if any new image uploaded to your S3 bucket is a cat or dog. In serverless architecture, the upload to S3 is the event trigger, which activates a function called `processImage()`, which allocates a server instance to execute the actual `processImage()` function, which is then deallocated when `processImage` finishes executing.

If you wanted to store your prediction result, you would need to go to a storage mechanism like BaaS, utilized as another API/function call, most likely outside of the instance that was just instantiated to execute `processImage()`. Otherwise, the result of `processImage()` would disappear at the end of the function's execution.

Diving further into how serverless architectures differ from traditional setups, consider a simple architecture without serverless for a photo sharing application.

A client makes a request on the internet, with a load balancer that appropriately forwards the request to a server (e.g., an EC2 Instance) that is managed by an autoscaler and contains code for various application functionality (find friends, searching for or posting photos, etc). The server then sends the results (e.g., a photo or relationship adding a friend) to a database for storage.







## The developer's role

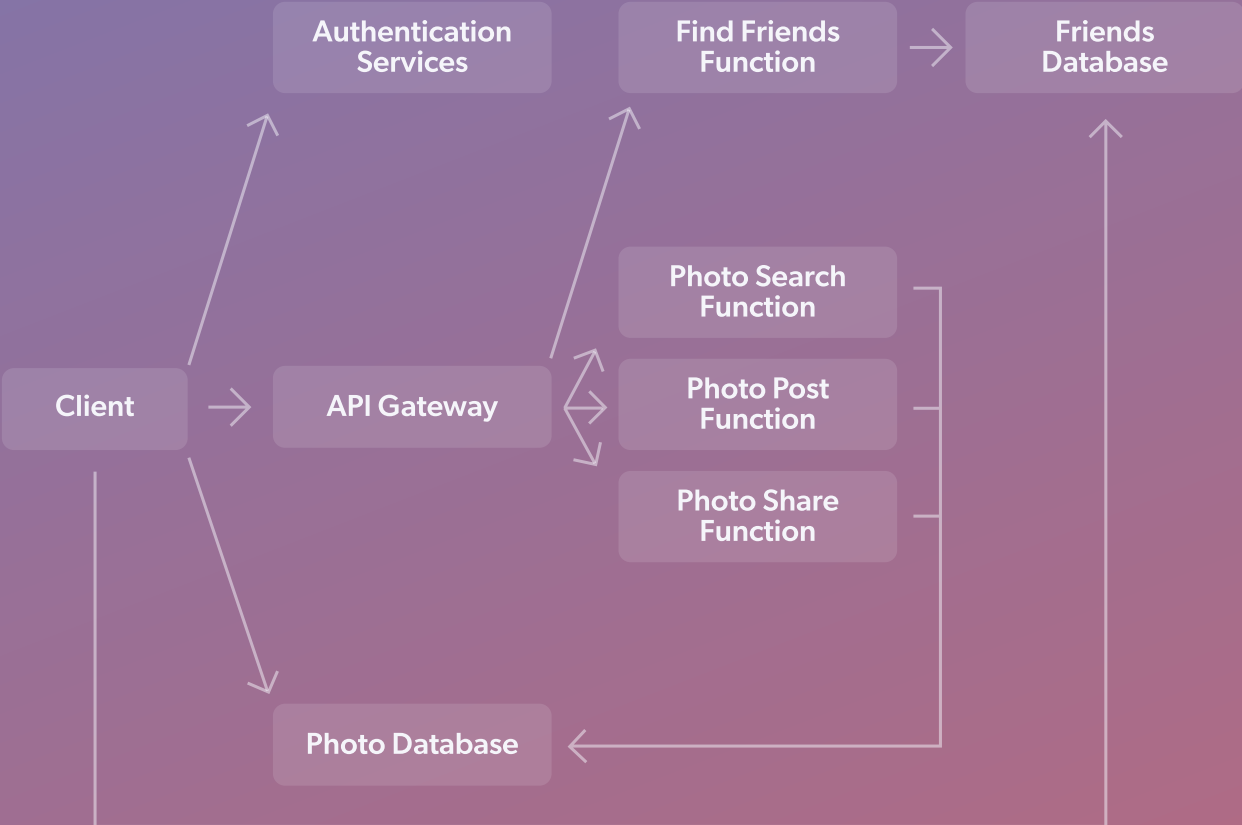
The developer has many tasks in creating and maintaining this architecture. They need to properly administer security updates across multiple environments, install packages, and maintain configuration files.

They also need to specify to AWS the parameters of the EC2s they would like to use, such as the number of cores for memory and CPU, and set rules on the Elastic Load Balancer regarding how to monitor environments for provisioning new servers and what thresholds to scale up or down. The server in this architecture handles a wide range of responsibilities, such as authentication and searching.

These tasks not only take time, but can also have a drastic effect on the performance of the product. Purchasing too many CPUs will drain a business's cash. On the other hand, having strict auto-scaling thresholds or purchasing server capabilities below your requirements will provide a poor customer experience.

# Basic architecture

Now, let's consider a very basic serverless architecture of our photo sharing application.



The first important difference to note is the decoupling of serverless architectures. The responsibilities of the server previously (e.g., authentication) have been delegated to a standalone managed microservice, while database functionality has been broken into disparate subsets that specific functions have access to.



DEGREE OF ABSTRACTION

## Applications

Web APIs

Event Data Processing

Serverless Applications

## Serverless

Cloud Functions

Object Storage

Key-Value Database

Big Data Transform

Mobile Backend Database

Big Data Query

Serverless Cloud Services

## Base Cloud Platform

VM

VPC

Block Storage

IAM

Billing

Monitoring

## Hardware

Server

Network

Storage

Accelerator

While the server was the central orchestration mechanism in our prior architecture governing all facets of the product, here the responsibilities are broken into constituent components responsible for a specific task in a microservice-oriented manner.

Instead of a load balancer, an API Gateway controls incoming requests and—based on the event activated—triggers a particular function in place of an EC2 instance.

You'll also notice that the client and functions are accessing the same databases. While it is not normally the case that all databases are accessible by both the backend and the client, it is more common in serverless architectures for some of the previously server side logic to be implemented on the client side instead (e.g., reading from a database to render content to a user).

Finally, since functions are instantiated upon demand, developers no longer need to make decisions regarding EC2 requirements and autoscaling threshold.

# Benefits of serverless

## Big cost savings

The single biggest benefit of serverless technology that makes it so popular is cost savings. As more companies adopt the cloud, there is a greater need to optimize cloud costs. In fact, optimizing cloud cost has been the most important initiative and active responsibility of IT teams for the last three years<sup>5</sup>.

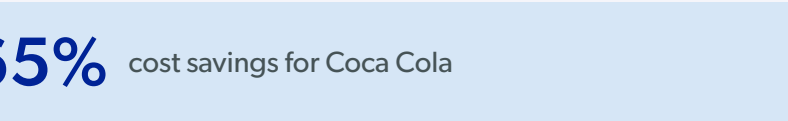
Before we go further, it is important to understand how serverless technology providers like AWS Lambda charge users. Since developers no longer have to worry about provisioning servers, they no longer have to pay for servers even when the servers are not running. Instead, they pay purely for function execution at a millisecond level, along with the memory allocated for the function upon being triggered.

Since compute and runtime fees are limited to actual usage, companies can experience significant savings with serverless.

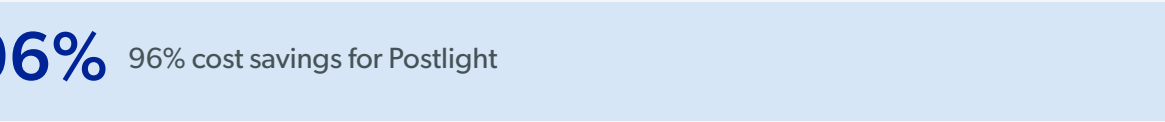
## Large scale examples

Companies like Netflix use AWS Lambda to serve seven billion hours of video every quarter. (You can imagine the difficulties of scaling up Netflix's architectures for Stranger Things releases without serverless.)

## Migrating to serverless pays off



**65%** cost savings for Coca Cola



**96%** 96% cost savings for Postlight

Cloud server providers are incentivized to offer such pricing models because they want to achieve business growth, maximize utilization, and provide efficient resource allocation. Making the cloud easier to develop upon ultimately attracts more customers and helps existing customers use more cloud services.

<sup>5</sup> **Source:** RightScale's 2019 State of the Cloud Report

# Improved developer experience

The second major benefit is an improved developer experience (in some areas at least—we'll touch more on this later).

Serverless makes API versioning trivial—just spin up a new function for a new API version. Horizontal scaling is now as simple as letting cloud providers manage application spikes and downtime.

Patching security updates is also simplified, as the application has been broken down into flexible and independent components. This not only allows for easier application maintenance, but also ease of deployment time and resources and greater flexibility in innovating and building upon serverless architectures.

## Reduced overhead

The third major benefit is a direct result of the first two. Companies no longer have to focus resources on managing infrastructure and deployment complexity. The operational headache is eliminated and companies can instead optimize upon product development and user experience.

## New marketplaces

Finally, a fourth major benefit is introducing a new marketplace based on functions (FaaS). Developers could offer a domain-specific function in the marketplace, which could be utilized by other app builders. The efficiency and feature richness of each function will drive its price and adoption.



# Barriers to adoption

## Overview

**“The simple stuff isn’t always easy. I wanted to do something very straightforward—to write a simple python function that would react to an API call and get information from DynamoDB. The function was about 3 lines of code...but Amazon has made this very painful...I got to the point of trying to do things directly with the Amazon stack that made me want to throw my monitor out the window.”**

*- Sam Newman, Instructor of Migrating Microservices to serverless—Understanding serverless Technology and Addressing serverless Concerns*

Serverless technology has a lot of promise and can even be seen as the natural direction of computing. After all, there is an increasing trend of developers off-shoring capabilities to third-party managed services so they can focus on product development. Which begs the question—why not do it for servers?

While the space continues to evolve and startups emerge around each of these areas, the very premise behind serverless creates a number of issues.

A Serverless.com survey reports that the largest barriers to entry include best practices, lack of tooling, startup latency, and lack of knowledge, while the largest challenges reported from respondents include expressibility, debugging, monitoring, and testing.



# Performance

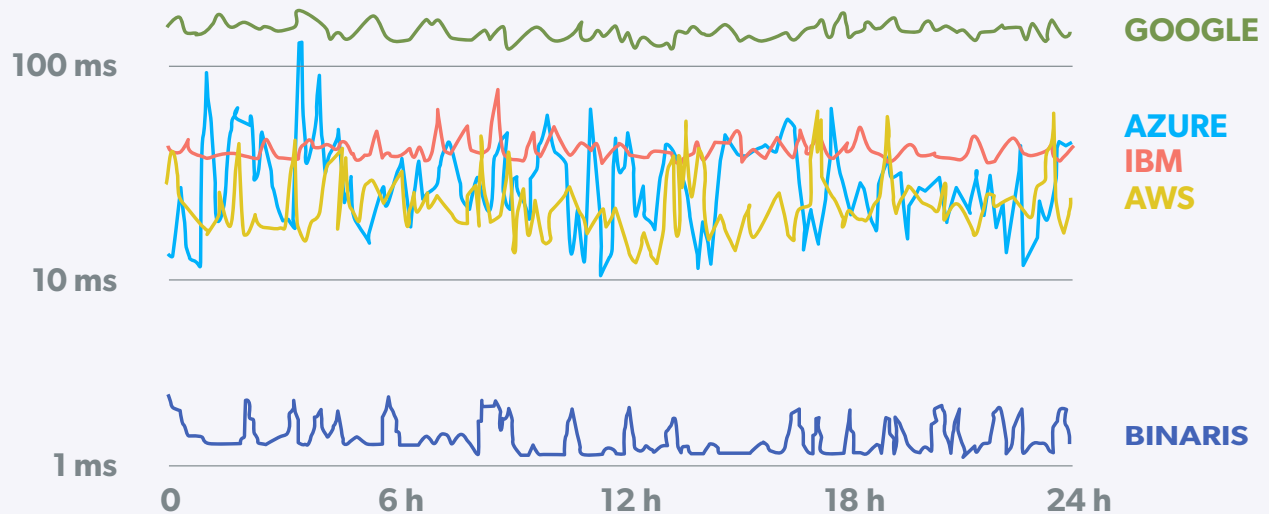
The most important consideration in adopting serverless technology is overcoming performance issues related to speed, overhead and parameter limitations.

## Unreliable performance

The first major issue of serverless is the fact that it can be very unpredictable or downright slow for a variety of reasons.

Reshuffle (formerly Binaris) notes that invocation responses from the cloud providers are guaranteed at the 99th percentile, meaning 99 out of 100 responses are guaranteed to be within a specific timeframe, which is not viable for use cases like gaming, ad bidding, AR, and industrial IoT that require consistent guarantees of responses within a set amount of milliseconds.

This is an active area of research in both academia and industry. Binaris also built an opensource tool called FaaSmark to measure function latencies across platforms, demonstrating significant variations in performance across major cloud providers.



Source: Reshuffle (formerly Binaris)

After measuring latency across cloud providers, there's approximately a 100ms latency difference between Google and Binaris. Providers Azure, IBM and AWS land somewhere in between for speed, but most have much wider ranges of variability.

6 Source: Binaris Blog: Serverless at Scale Serving StackOverflow like Traffic

# Stateless and communication overhead

Stateless components must interact with stateful components to persist information, which can introduce latency and complexity.

In traditional architectures, application components that must speak to one another can be co-located within the same server or rack or instance or use optimized network protocols to reduce latency. Therefore, the co-location of both compute and data in traditional architectures can deliver optimization that is more difficult with serverless.

A serverless architecture composition is merely a set of functions that deliver desired semantics, so there isn't a guarantee they are co-located. This actually increases communication overhead.

Currently, inter-component communication is generally done via HTTP, RPC, or some Shared Message Queue, which is an order of magnitude slower than shared memory or local messages on the same compute server.

## Loss of control

By using serverless, developers are limited to a small number of configuration parameters—namely memory size—with far less control over JVM or operating system runtime parameters, leading to performance issues.

Tests have been done running on FaaS providers, where identically configured functions have drastically different performance characteristics as a result of cloud providers altering scheduling priorities and resource allocations in response to demand. Language runtimes also differ drastically in cold start time, with Java often several seconds longer than Python.

# Cold starts

Perhaps the most frequent complaint amongst the serverless opposition is the idea of Cold Starts.

When an event triggers, the FaaS provider (e.g., AWS Lambda) spins up a container, loads all code and dependencies in memory, and then runs the function inside to eventually deallocate the container. In reality, this container is “kept alive” and sits idle for a period of time awaiting additional requests (minutes or hours).

The instantiation of the container from the first function called is referred to as a “Cold Start”, while proceeding function executions occur while the container is “Warm”. The problem here is that execution of cold functions are drastically slower than warm functions—many times on the orders of seconds when compared to milliseconds for warm functions.

## How developers deal

There are two ways that developers attempt to deal with the problem of cold starts today, hacks and suboptimal.

The first is to repeatedly call their function every few minutes to continually keep their containers warm. However, the problem with this is that there are concurrency limits stipulated by the major cloud providers to prevent the number of instances serving requests from becoming too high. Furthermore, these warm containers maintain their connections to external databases, and thus, the databases themselves can suffer from concurrency issues from warm functions.

The second method is to increase the memory allocation parameter for your function, as AWS allocates CPU to functions in proportion to the memory. For example, a function allocated 256MB of RAM will receive 2x the CPU processing from AWS than a 128MB function. As a result, developers will allocate more memory than necessary to functions in order to bolster performance. However, this too is a suboptimal solution. While allocating 3GB of memory to a function requiring 128MB may mitigate cold start latencies, it does not help reduce costs for the much more numerous warm executions occurring.

# Expressibility

While serverless technology has come a long way, it doesn't yet have the fine tuned controls of some more established technologies. This results in higher latencies and inefficient data dependencies.

## Fine grained storage and coordination operations

Any non-trivial application ultimately always needs fine grained state sharing and a means for task coordination or sequencing.

While more cloud storage services offer scalable long-term storage, they tend to be ill-equipped to handle fine-grained access needs, resulting in high access cost and high latencies.

Storage services that might help to manage or preserve state don't come with notification or coordination services out of the box, forcing application developers to implement a notification/rendezvous system using services. This adds both latency and complexity.

## Data dependencies

Serverless platforms today do not inherently provide a mechanism to express data dependencies between functions, which would assist the serverless service provider on resource allocation. As a result, this can lead to suboptimal infrastructure allocation and communication overhead for the application.

## Function cost analysis tooling

With serverless, developers do not have to think about latency, scalability, and fault-tolerance from a server infrastructure point of view.

Instead, developers must concern themselves with functions, carefully considering execution time and the resources each uses. Unlike various static analysis tools that exist today, there is a dearth in powerful tooling that ties one's code and runtime execution through a single lens to provide insight into cost. This is important because you still pay for server time even when an execution thread is waiting.

# Operational monitoring and testing

While serverless certainly helps mitigate certain parts of the developer experience—namely removing infrastructure maintenance—it does not help in many other areas.

## Testing

Local testing is a foremost problem for serverless today.

Since infrastructure is abstracted away in a serverless setting, incorporating production-like error handling, logging, performance, and scaling characteristics is notoriously difficult in a local environment.

Testing remotely is also inherently difficult due to the unpredictable nature of serverless container instances. Testing is only limited to individual functions, as opposed to overall application testing.

## Debugging

There are few services available to remotely debug serverless compute components like traditional run-time debuggers with line-by-line stepping. Unlike traditional architectures, one cannot simply log into an EC2 instance to discover what went wrong.

## Monitoring & logging

Serverless monitoring and logging services require more work. AWS Lambda/CloudWatch functionalities are quite poor, and monitoring issues and metrics across a distributed system of hundreds of functions is a pain point today.

As a result, many developers use AWS console or a CLI with significant code written to manage functions, since stitching together various cloud provider components and services requires substantial work.

# Deployment

Today, it is difficult to orchestrate large-scale serverless applications due to being composed of many individual components.

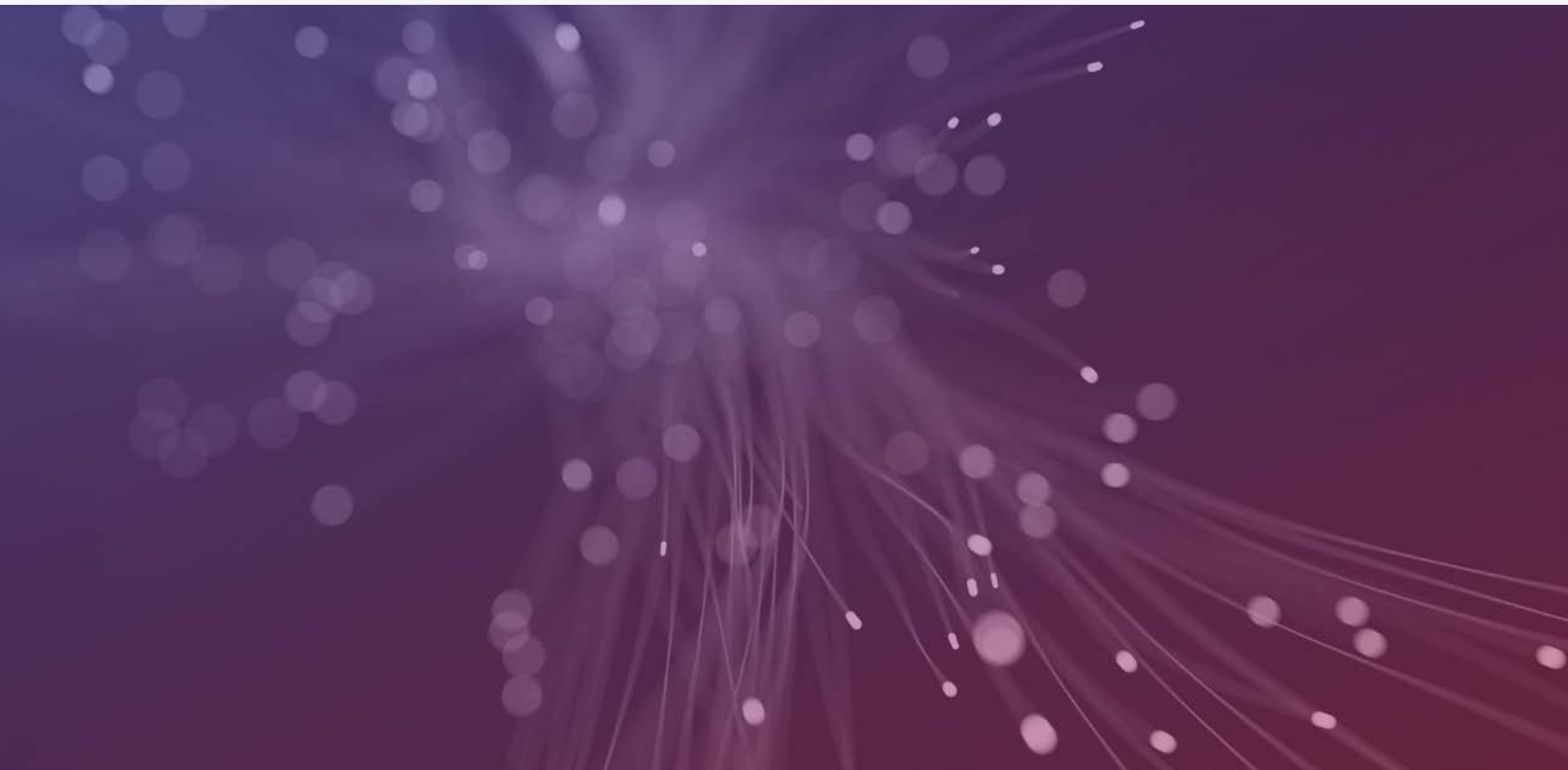
Even deploying a simple “Hello World” function requires bundling dependencies, establishing connections to API Gateway and stateful databases, managing config files, and numerous other tasks.

An illustration of the difficulties in deployment for serverless revolves around package dependencies. Lambda functions are zipped into a file and deployed to the FaaS provider, where the FaaS makes the functions accessible via HTTP requests from triggers.

As a result, users must download and configure all dependencies before delivering them to a server, as opposed to relying on a dependency manager. Furthermore, many NPM packages are not compatible with serverless architectures.

## Vendor lock-in

As is the case with cloud providers, employing a specific serverless architecture may lock-in customers with a particular vendor, or result in substantial code rewriting. Enabling hybrid serverless models across multi-cloud environments is still a work in progress.





# Security and privacy

As with any new technology, there are many concerns from the security community. Examples include: event-data injection to functions, applying authentication to hundreds of functions, function permission and role management, and inadequate function monitoring and logging tools. Israeli Security Firm PureSec found 1 in 5 of 1,000 serverless applications audited held critical vulnerabilities<sup>7</sup>.

The context for privacy considerations is even trickier. Unlike security attacks where the crux is to manipulate intent, in the case of privacy even the context of data or the sequence of function calls might reveal sensitive private data. For example, in the case of a health-care monitoring product, a sequence of function calls may reveal certain predilections about the patient.

## Runtime limitations

Each cloud provider has limits on the amount of time any function can execute for before timing out, as well as a memory limit developers can request for a given function.

	<b>AWS Lambda</b>	<b>Microsoft Azure Functions</b>	<b>Google Cloud Function</b>
Runtime Limit	15 minutes	10 minutes	9 minutes
Memory Limit	3008 MB	1536 MB	2048 MB

<sup>7</sup> **SOURCE:** Network World: One in five serverless apps has a critical security vulnerability

# Confusing costs

While serverless can lead to significant cost savings, it can also be prohibitively costly when not used appropriately and difficult to understand how to optimize due to charges made by a number of invocations, duration of invocations, and size (memory) allocated for each invocation.

## Duration

AWS charges in increments of 100ms—if a function runs for 102 ms, AWS will charge for 200ms of runtime. Lambda also charges for function duration even if the function is waiting for another microservice to complete IO, and thus functions calling functions can create cascading wait times.

## Memory

Memory scales proportionally to cost—a 128MB function is half the cost of a 256MB function. However, CPU resources are also allocated proportionally based on memory size.

Users must employ optimization strategies on a per function basis; for example, a 110ms / 128MB invocation cost is rounded to 200ms, but increasing the memory to 192MB could speed up I/O operations and reduce execution time to below 100ms to save 25% per invocation. Thus, figuring out what memory size to use for each function becomes a time intensive and costly operation.

## Additional costs

There are additional costs built into the many microservices associated with ensuring lambda works, such as API Gateway, CloudWatch, background processors like SNS/SQS/Kinesis, data transfers, etc.

Cloudwatch is a required service that automatically logs start, end, and report messages for each function, creating costs even if one uses an alternative aggregation service across all microservice logging.

# Current market landscape

The serverless market map below consists of startups, open source packages, and large company offerings in areas with the greatest needs and resources being invested.

## PLATFORMS



## TOOLING

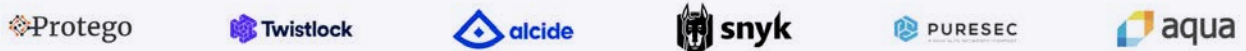
### Deployment



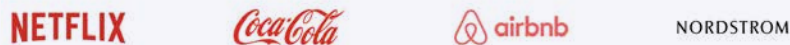
### Monitoring, logging, and debugging



### Security



## USERS\*



To better illustrate the current landscape, we have featured one startup from each category in further detail below. Our choice of which startups to highlight is NOT reflective of our evaluation of these companies as investments or as leaders in their categories, but to provide a better picture of the types of startups emerging in the serverless space.

\*We included the "Users" subsector for completeness—not as an area of emerging serverless startups.

## Platforms



## Platforms

# Refinery.io

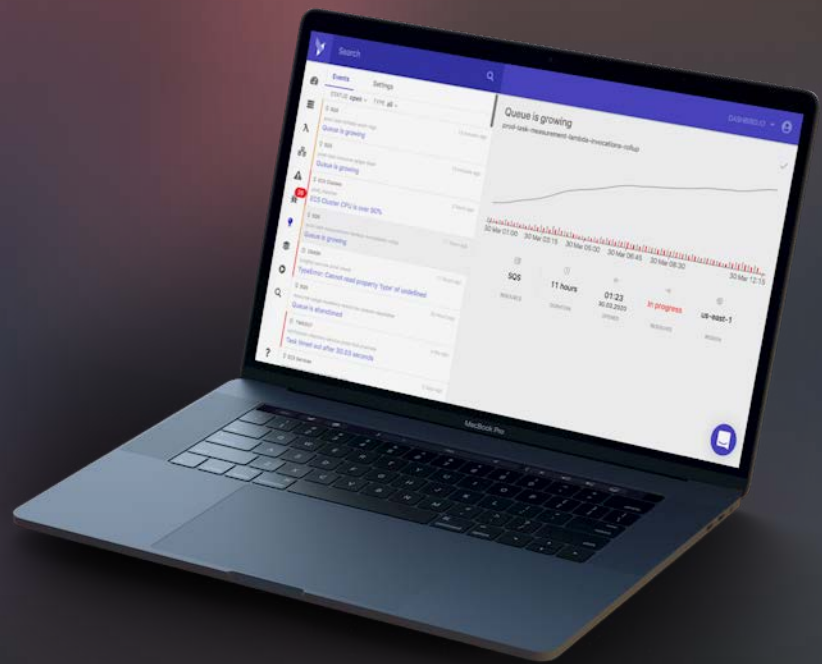
Refinery.io allows developers to create serverless applications by abstracting away the underlying infrastructure pertaining to serverless.

Using a drag and drop editor with underlying primitives, developers can link together Code Blocks, API Endpoints, and API Response Blocks to easily create functional web endpoints built on serverless.

With Refinery, developers no longer need to deal with the complexities of coding with lambdas or to even understand Serverless at all, but use abstractions of various cloud services, lambdas, API Gateway, jobs, and queues.

In addition to easily creating serverless functions, Refinery's platform aims to tackle several other common grievances of using lambda today.

- Charges for compute used and not idle servers waiting for lambdas to execute
- Serverless Map Reduce to distribute workload across thousands of servers
- Visual debugging and logging systems to follow execution flow of deployed services



## Monitoring, logging, and debugging

# Dashbird

Dashbird has over 1000 companies using its product and 3,400 AWS accounts connected to its service. The product is only compliant with AWS Lambda and Java/Node.js/Go/Python, though it's a popular offering.

Dashbird provides a product for monitoring operations across serverless deployments. Today, it is difficult to understand lambda function errors and microservice malfunctions in real-time in such distributed architectures, and many developers believe AWS CloudWatch has poor log-search and reporting capabilities.

Dashbird can be set up in minutes with zero code changes by deploying a preconfigured CloudFormation template and provides:

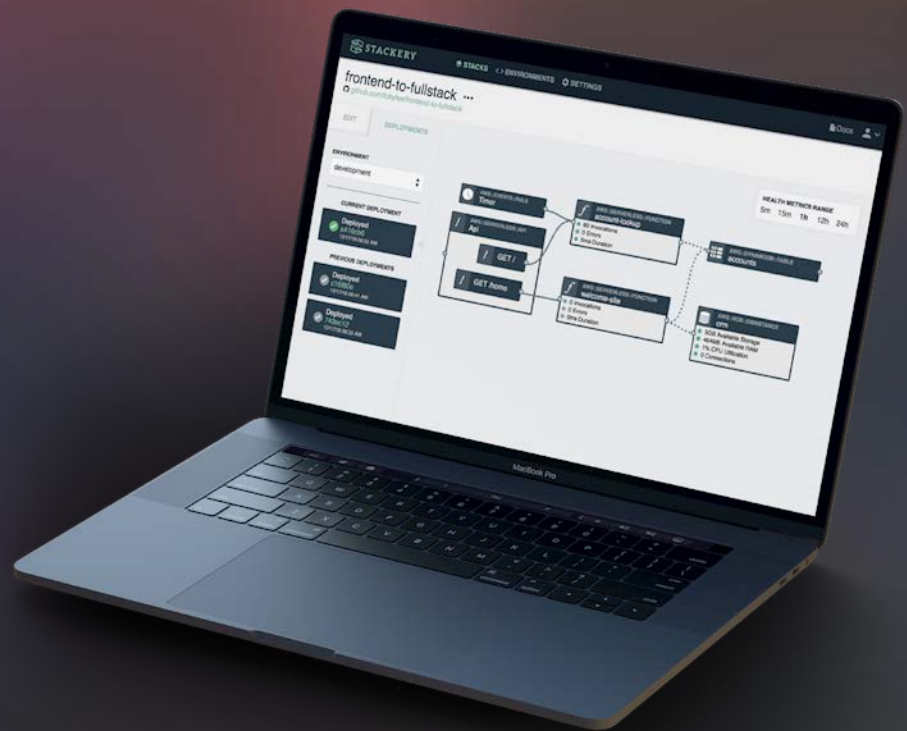
- Visualizations such as total invocation counts, errors, cost, billed durations, and memory utilization, as well as Cold Start detection and impact analyses

- Alerts for anomalies and potential issues (e.g., functions approaching memory limits, increased # of timeouts, approaching timeout limits, etc.)

- Detailed tracing analyses and stack traces for function errors; searching capabilities for logs, metrics, and traces, including elastic search for AWS Lambda logs

- AI based recommendations for memory setting changes to speed up systems and function optimizations to cut costs (e.g., detecting over or under provisioned lambda functions)

- Integration with Slack, email, and other systems to notify of alerts in real-time



## Deployment

# Stackery

Stackery provides a combined visualization tool and CLI to build and deploy serverless architectures consistently across development, testing, and production environments.

Its visualization dashboard is its core differentiator from competitors in the space. It provides a drag and drop tool to create cloud side development environments and functions, and link them to events and stateful databases.

It also emulates the architecture locally in an IDE for increased development speed, where it manages service-to-service permissions and configurations between both local and cloud resources.

Accelerated development by enabling iteration in sections. Tools accelerate development by up to 60x with rapid deployment

Visualization of complete architecture and local cloud-testing capabilities

Collaboration tools including rollback protection, automated build packaging, GitHub integration, and partnerships with serverless monitoring tools like Epsagon





## Security

# Protego

Protego provides security for serverless applications from deployment to runtime. The platform can get up and running in 20 minutes, and supports the major cloud providers and functions written in Node.js, Python, and Java.

Existing security solutions focus on a small number of entry points, whereas serverless infrastructures expose hundreds of functions as entry points.

Protego's web-based SaaS application:

- Continuously scans an application to create a model of the application functioning normally, and uses deep learning to surface threats, anomalies, and malicious attacks
- Examines architectures for potentially dangerous interactions between functions and microservices; provides a comprehensive view of serverless ecosystem, visualizing all inputs, triggers, and risk areas
- Runtime protection mechanism to inspect and filter function-input data
- Automatically detects configuration risks and generates least-privilege function permissions during CI/CD
- Helps manage vulnerabilities of 3rd party libraries, and adapts security level applied to serverless resources to minimize resource consumption while ensuring protection
- Visualizations of data points relevant to a particular security event with audit trails
- Automatically detects and mitigates SQL Injections and anomalous activities (e.g., calling external destinations or sub-processes)

# Closing thoughts

We're still in the early innings of serverless technology. While companies like Netflix, AutoDesk, Nordstrom, Thomson Reuters, and others are early adopters, there are many companies not using it (or entirely unaware of its existence) and others who are using it in production for only small, non-critical parts of their applications.

Even so, serverless has come a long way since AWS Lambda launched in November 2014. There are more serverless providers than ever before and the space is continuing to evolve at a rapid pace.

In the same way that the rise of cloud computing enabled hundreds of successful startups to be built upon the major providers' architectures and enable ease of usage and enhanced functionality, we're seeing startups built on top of AWS Lambda Functions, Microsoft Azure Functions, and Google Cloud Functions.

While there are interesting startups in all of the categories described previously, we at Unusual Ventures believe that noteworthy areas of opportunity include tackling the issues of security and improved storage options that allow for reduced latency and stateful workloads on serverless, as well as solutions that assist developers in mapping out serverless architecture. We believe the serverless movement is just getting started and that it's going to be big. Serverless computing could become the de-facto programming model in the next stage of the cloud era.

The serverless community is steadily growing, with popular solutions like Serverless Inc at 30,000+ Github stars, and once the main barriers to entry are mitigated—namely ease of deployment, enhanced monitoring and security tools, reduced latency, and additional storage solutions—serverless will see a surge in adoption.

At Unusual, we anticipate a migration over time in the enterprise toward distributed architectures and increased adoption of serverless in the future, and we are excited to be supporters of the serverless movement.

# Contact

If you think serverless is interesting, are working on unusual tech (serverless or otherwise), or simply want to chat, we want to hear from you. You can follow us on Twitter, Connect on LinkedIn, or just shoot us an email.



**UNUSUAL**  
VENTURES

**Jordan Segall**

@jordan\_segall  
jordan@unusual.vc

**Abhi Sharma**

@abhisharma\_b  
abhi@relyance.ai