Master Thesis

Master of Science (M.sc)

Department of Tech and Software

Major: Software Engineering (SE)

Topic: Evolving React Architectures - The Scalability and Cost Advantages of the Serverless

Author: Aiswarya Sajeev

Matrikel-Number: 57338808

First supervisor: Prof. Dr. Rand Koutaly

Second supervisor: Dr.Souad El Hassanie

Submitted on: August 19, 2025

University of Applied Sciences Europe
Iserlohn · Berlin · Hamburg

ENSTÄNDIGKEITSERKLÄRUNG / STATEMENT OF AUTHORSHIP

Sajeev	Aiswarya
amily Name	Vorname | First Name

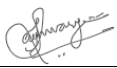
57338808	Evolving React Architectures : The Scalability and Cost Advantages of the serverless
ummer | Student ID	Titel der Examsarbeit | Title of Thesis

chere durch meine Unterschrift, dass ich die hier vorgelegte Arbeit selbstständig verfasst habe. Ich habe zu keiner anderen als der im Anhang verzeichneten Quellen und Hilfsmittel, insbesondere keiner nicht en Onlinequellen, bedient. Alles aus den benutzten Quellen wörtlich oder sinngemäß übernommen Teile b Textstellen, bildliche Darstellungen usw.) sind als solche einzeln kenntlich gemacht.

egende Arbeit ist bislang keiner anderen Prüfungsbeh.rde vorgelegt worden. Sie war weder in gleicher noch her Weise Bestandteil einer Prüfungsleistung im bisherigen Studienverlauf und ist auch noch nicht publiziert. Druckschrift eingereichte Fassung der Arbeit ist in allen Teilen identisch mit der zeitgleich auf einem elektronischen medium eingereichten Fassung.

signature, I confirm to be the sole author of the thesis presented. Where the work of others has been consulted, ly acknowledged in the thesis' bibliography. All verbatim or referential use of the sources named in the aphy has been specifically indicated in the text.

sis at hand has not been presented to another examination board. It has not been part of an assignment over my f studies and has not been published. The paper version of this thesis is identical to the digital version handed in.

Berlin, 21-08-2025	
rt | Date, Place	Unterschrift | Signature

University of Europe for Applied Sciences

Declaration on the use of generative Artificial Intelligence (AI) systems

Sajeev
Name | Family Name

Aiswarya
Vorname | First Name

57338808
Matrikelnummer | Student ID Number

Master Thesis 2025
Titel Prüfungsarbeit | Title of the exam

I have used the following artificial intelligence (AI)-based tools in the creation of my work:
1. ChatGPT
2. Gemini Pro
3. Quillbot

I further declare that

☑ I have actively informed myself about the performance and limitations of the above-mentioned AI tools,
☑ I have marked the passages taken from the above-mentioned AI tools,
☑ I have checked that the content generated with the help of the above-mentioned AI tools and adopted by me is actually accurate,
☑ I am aware that, as the author of this work, I am responsible for the information and statements made in it.

I have used the AI-based tools mentioned above as shown in the table below.

AI-based support tool	Usage	Parts of the work affected	Remarks
ChatGPT (Open AI GPT – 5)	Assisted in structuring thesis outline, suggesting literature search strategies, drafting methodological descriptions, and summarising test results. Used as a research assistant to improve clarity of explanations	Introduction, Literature Review, Methodology, Results & Discussion (supportive role only)	Content was critically reviewed, edited, and rewritten by me. AI was used for support, not for final argumentation
Quillbot	Paraphrasing of sentences to improve readability and reduce similarity in plagiarism checks	Literature Review and Introduction	Used only for language refinement. Original meaning and citations were preserved by me.
Gemini (Google)	Cross-checking facts, terminology, and ensuring consistency in technical definitions	Mainly Literature Review and Discussions	Served as a validation/checking tool; no direct text copied

Berlin, 2025-08-21, Aiswarya Sajeev

Abstract

The design and delivery of contemporary programs has been completely altered by cloud computing, with serverless computing emerging as a particularly significant approach. Serverless computing abstracts away infrastructure management, allowing developers to focus on application logic while the cloud provider takes care of resource provisioning, fault tolerance, and scaling automatically. This thesis uses a React-based to-do application hosted on Amazon Web Services (AWS) as the experimental case study to look into how well serverless architecture works for creating scalable and economical full-stack apps.

Using a design science technique, the study combines the creation of prototypes with empirical testing and assessment. AWS Lambda, API Gateway, DynamoDB, Cognito, and S3 were utilised in the system's implementation, and deployment automation was supported by AWS Serverless Application Model (SAM). Dotcom-Monitor for end-to-end monitoring and Artillery for load testing were used to evaluate performance. Latency, throughput, scalability under different traffic scenarios, error rates, and operational cost are important indicators for assessment. The application grew flawlessly to more than 50,000 requests per second without any issues, according to the results, and achieved sub-100 ms latencies under moderate loads. Although there were some cold start delays, they had little overall impact on prolonged workloads.

In contrast to conventional server-based or containerised models, the results demonstrate that serverless architectures offer excellent scalability and notable operational simplicity, making them extremely useful for workloads with variable or unpredictable demand. The study does, however, also point out trade-offs, such as reliance on external monitoring tools, decreased system observability, and greater relative costs for situations with continuous high traffic. All things taken into account, this thesis highlights the advantages and disadvantages of serverless computing in full-stack development and makes the case that hybrid models, which combine serverless with conventional cloud or container solutions, might offer the best course of action for applications that are latency-critical and cost-sensitive.

Keywords: Serverless Computing, AWS Lambda, React, Scalability, Latency, Cost Analysis, Cloud Applications

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Background

Over the past century, Web application has undergone huge change from using monolithic server application or architectures to microservice architectures (ref Figure 1). While monolithic architecture was easy to maintain and under one roof, world is adapting to more modular, scalable and event driven microservices. As a result of this evolution, serverless computing has become a powerful paradigm that frees developers with the tasks of managing servers so they can work entirely on producing code and application logic.(Baldini et al., 2017). At the same period of time, component-based tools like React.js have grown as the de facto standard for creating interactive and consistent user interfaces, greatly improving frontend development.

Containers or Virtual Machines(VMs) are the places where backend services for traditional web applications are usually maintained at. Provisioning, scaling and patching such specific infrastructure management is required for these implementation. Even though platforms like Docker and Kubernetes have made deployment pipelines more efficient, they still involve an important amount of DevOps skills and work (Jonas et al., 2019a). On the other hand, serverless architectures, especially those built with Function-as-a-Service (FaaS) platforms like Google Cloud Functions, Azure Functions, and AWS Lambda, allow developers to respond to events with code without having to worry about supporting the underlying infrastructure (McGrath and Brenner, 2017).

Three main advantages of the serverless paradigm are enhanced development productivity, lower operating costs, and automatic scaling(Hellerstein et al., 2018). Because of these advantages, it is useful for creating full-stack applications, where the frontend and backend can be expanded independently, asynchronously called and loosely connected. Notably, serverless computing complements modern methods of development such as event-driven microservices, Infrastructure-as-Code (IaC), and Continuous Integration/Continuous Deployment (CI/CD) (Spillner, 2019).

At the same time, the shift to Jamstack architectures – which are built on JavaScript, APIs, and Markup – has encouraged a wider ecosystem of applications that are static first and dynamic on demand (Biørn-Hansen et al., 2019). In this case, React is the frontend layer which is frequently packaged and static hosting services are made available through static hosting services like Vercel, Netlify, or Amazon S3. Backend logic were supplied using the serverless APIs. Smooth scalability, security by design, and near-instant global performance are the features provided by this architecture.

Howere, there are new difficulties when incorporating serverless architectures into full-stack applications. When implementing production-grade applications, factors like cold-start delay, function state management, security isolation, API orchestration, and debugging complexity need to be taken into account.(Hendrickson et al., n.d.). Additionally, comparable evaluations are challenging due to the absence of industry-wide standards for function design, observability, and performance benchmarking. (Eismann et al., 2022; Shillaker and Pietzuch, n.d.)

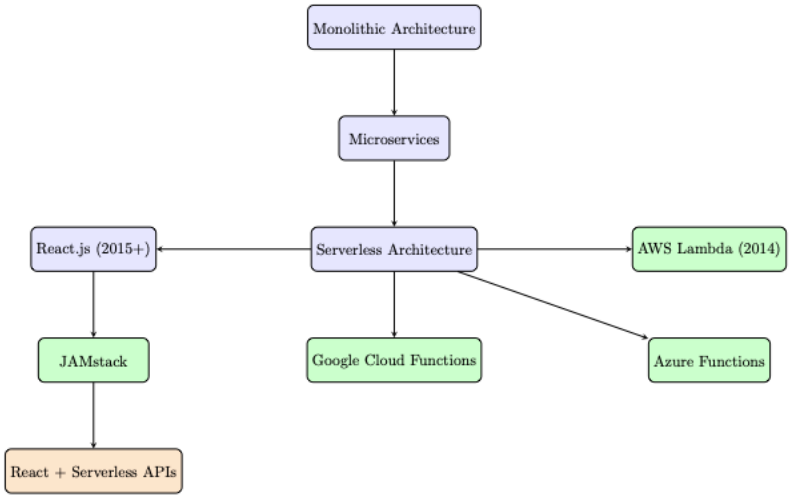


Figure 1 Evolution from Monolithic to Serverless and React Integration

1.1.1 The Modern Web Application Landscape

The digital eco system has an unlimited demand for web development which are not only rich in functionality and interaction but also exceptionally performant and greatly scalable. In this landscape, React has risen as a ruling Javascript library for building sophisticated user interfaces. Industry giants such as Meta, Netflix and Airbnb has adopted it to power the key components of their platform, at a global scale. The foundational principles must be highly credited for the success of React: reusability and modularity is encouraged by a component-based architecture and expensive direct Document Object Model (DOM) manipulations is minimized by a virtual DOM that offers an efficient rendering mechanism. These features have helped developers to build complex, state-driven applications, still there was a continuous and intense evolution in the architectural patterns for deploying and rendering these applications.

1.1.2 The Architectural development of Frontend Rendering

The principle of Client-Side Rendering (CSR) determined the early wave of modern Single-Page Applications (SPAs), often built with libraries like React. In a CSR model, a considerable Javascript

bundle is downloaded and executed to render the entire user interface dynamically, after the server delivers a minimal HTML document to the browser. This approach introduced critical limitations that created a new phase of architectural invention, while enabling fluid, app-like interactivity after the initial load. The main drawbacks of CSR were twice as much: poor initial load times caused by notable performance bottlenecks as calculated by Core Web Vitals (CWV) like First Contentful Paint (FCP) and Time to Interactive (TTI); and serious challenges for Search Engine Optimization (SEO), as a large empty HTML shell was experienced by web crawlers, failing to index the dynamically rendered content.

As a straight response to this imperfections, server-centric rendering patterns were developed by the frontend community. Server-Side Rendering (SSR) came out as an impactful solution, where a fully rendered HTML page was generated by the server for each request, immediately providing meaningful content to both users and search engines. The core SEO shortage and perceived performance of CSR were resolved and greatly improved. Simultaneously, Static Site Generation (SSG) presented an even more high performing alternative by pre-rendering all pages to static HTML files at build time, which can be delivered immediately from a global Content Delivery Network (CDN). Meta-frameworks, most particularly Next.js, have been crucial in promoting and refining the implementation of these patterns within the React ecosystem.

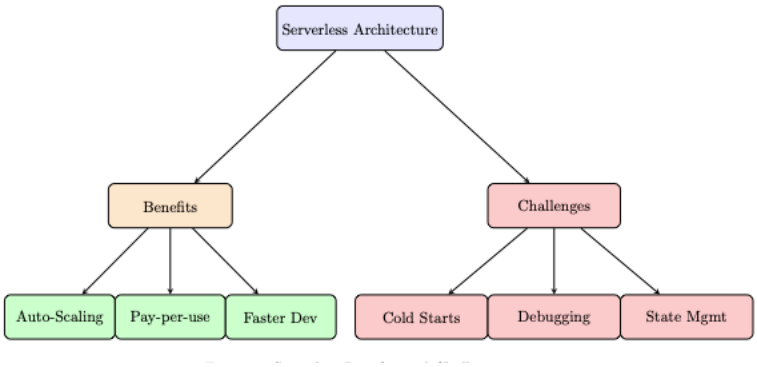


Figure 2 Serverless Benefits and Challenges

This evolutionary path has continued to enhance the balance between static performance and dynamic capability. The need of full site rebuild is replaced by hybrid models such as Incremental Static Regeneration (ISR) which extends SSG by granting static pages to be updated at intervals or on-demand. The most latest progress, React Server Components(RSCs), enabled components to execute completely on the server, denoting a model shift, as a result eliminating their Javascript impression on the client and further enhancing client-side performance. A crucial architectural trend can be revealed from the advancement from client-heavy to server-centric rendering: an intentional and consistent shift of computational load from the variable environment of the user's browser back to the regulated, powerful environment of the server. This planned migration towards

server-side logic paves the way for a natural merging with the parallel advancements in cloud infrastructure.

1.1.3 The Parallel Rise of the Serverless Paradigm

The emergence of serverless computing, the revolution which was occurring in the backend infrastructure, concurrently with the evolution in frontend architecture. (Jonas et al., 2019a). This model doesn't mean that the servers are absent, but rather the complete concept of server management, transferring the operational responsibilities of provisioning, scaling, patching, and maintenance from the developer to the cloud provider (Jonas et al., 2019a). The idea of how applications are built and deployed is basically altered by this paradigm which is defined by a set of core principles.

First, serverless computing is predominantly event-driven, operating on a Function-as-a-Service (FaaS) model (Van Eyk et al., 2018). In this model, in response to specific triggers, such as an HTTP request, a database update, or a file upload, applications that are disintegrated into small, stateless functions are executed. Second, automatic and considerably infinite scalability is being offered; to meet the demand, resources are being dynamically allocated by the cloud platform, scaling from zero to thousands of parallel requests and again to zero without any manual intervention (Anselmi et al., 2025). Third, groundbreaking pay-per-use cost model is introduced, where the precise resource consumption during the execution is being calculated for billing, usually measured in milliseconds, thereby the cost of unused server capacity is eliminated. (Jonas et al., 2019a).

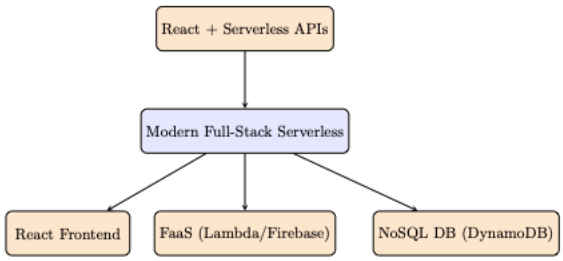


Figure 3 Modern Full-Stack Serverless Pattern

1.1.4 The Convergence and the Research Problem

Its more appropriate to say the development of cloud infrastructure and Reach architectures are strongly overlapping instead of a separate phenomena. An increasing amount of contemporary React meta-frameworks, such as Next.js and Remix are "serverless-native," meaning that their architectures are inline with the serverless paradigm. For example, SSR

and ISR like rendering strategies map perfectly onto the on-demand, event-driven compute provided by FaaS platforms, and API routes inside these frameworks are often made to be deployed as separate, isolated serverless functions.
Unprecedented levels of cost-effectives and scalability can be expected as a result of this convergence. In spite of the widespread promotion of the synergistic potential, there are important technical obstacles that must be overcome before these advantages can be identified in practice. Performance variability and the “cold start” issue – a noticeable lag added when a function is performed for the first time after a period of inactivity- are two complications that serverless architectures bring with them (Gao, 2024; Joosen et al., 2025). These performance issues have been brought to light by academic research, which shows that external service calls and trigger mechanisms frequently dominate end-to-end latency in serverless apps (Scheuner, 2022) and that performance can differ by up to 338% between the similar function executions (Wang et al., 2023).

Therefore, the main research issue that this thesis attempts to address is the following: Although the combination of serverless deployment models and developing React architectures presents a strong value proposition for higher scalability and cost savings, a thorough analysis is necessary to weigh these advantages against the operational difficulties and performance trade-offs that are unavoidable. The thesis will implement a mixed-method research strategy to address this, intregrating actual data from a specially designed experimental application with a comprehensive literature review. A thorough assessment of the important factors, such as cost-effectiveness(Total Cost of Ownership) and performance metrics (latency, throughput, and cold start times).

1.1.5 Significance and Contribution

The research’s conclusions can be a great benefit to both the academic community and industry practitioners. A thorough, fact-based framework for considering on web application architecture is being offered by this thesis to the developers, engineering leads and software architects. It gives a useful advice on creating systems that are both efficient and profitable by analysing the trade-offs between various rendering techniques and deployment patterns. This study additionally gives the idea about cloud computing, distributed systems and software engineering for the academic community. It provides an organized analysis of a new approach in the modern application development and lays the groundwork for the further studies into cost modelling, optimizing the performance and developer experience in serverless enviroments by synthesizing and critically analysing a quickly evolving technological intersection.

1.2 Problem Statement

The capacity of the conventional server-based architectures, especially in settings where quick deployment and agility are important, is surpassed by the need for cost-effective, highly scalable online applications. Traditionally, a combination of virtual machines or container orchestration solutions is used to deploy full-stack applications, which include both client and server layers. However, this approach frequently results in operational overheard and scalability problems (Wang et al., 2018).

The majority of current research either focuses on frontend optimization or backend scalability (such as FaaS runtime performance) separately, without fully exploring their integration in a full-stack environment, despite the fact that serverless computing presents a viable alternative (Van Eyk et al., 2018). In reality, there aren't any actual studies that assess the applications' overall maintainability, cost effectiveness, and real-world performance, although serverless APIs and microservices are commonly interfaced with React-based frontends.

This gap is especially urgent for startups and small businesses where international distribution and production-ready systems are needed with predictable prices and low operational complexity. When it comes to highlighting trade-offs between different serverless settings, function granularities, and integration techniques with frontend frameworks like React, developers sometimes lack evidence-based design patterns or comparison studies (Kaffes et al., 2019).

Therefore, the primary issue is the absence of extensive empirical and architectural analysis about the use of serverless architecture in creating full-stack scalable and reasonably priced React apps.

1.3 Research Aim and Objectives

This study seeks to close the gap between theory and practice by methodically examining the use of serverless architectures in full-stack web development with React. It can be said that performance, scalability, cost-effectiveness, and maintainability is the major goal for modern online applications while determining how serverless models may be designed, implemented and optimized.

Specific objectives include:

- To assess serverless backends' performance attributes (such as latency, cold starts, and throughput) under various loads.
- To calculate and contrast serverless installations' running costs with those of conventional architectures for full-stack apps.
- To determine the architectural approaches and patterns that support serverless React apps that are scalable and maintainable.
- To use React and a serverless backend (such as AWS Lambda + API Gateway + DynamoDB) to create and test a proof-of-concept application.
- To evaluate serverless full-stack application developments integration processes, tooling, and developer experience severely.

1.4 Scope and Limitations

This study is bounded by the following constraints:

- **Technology Scope**: The frontend will use React.js, while the backend will be leveraging AWS Lambda, API Gateway, DynamoDB, and S3 for static hosting.
- **Functionality Scope**: A CRUD system will be simulated by the implemented application with user authentication, data persistence, and third-party API consumption.
- **Performance Metrics**: Only backend latency, cold start duration, and monthly cost estimates will be considered; frontend performance optimization is out of scope.
- **Security**: While serverless security concerns (e.g., privilege escalation, IAM configuration) will be acknowledged, a comprehensive security audit is beyond this study's remit.

1.5 Thesis Organisation

This thesis's remaining sections are structured as follows:

- Chapter 2: Literature Review – Talks about the existing research on serverless computing, scalability, cost-efficiency, and architectural best practices.
- Chapter 3: Methodology – Describes the research design, prototype development, evaluation metrics, and data collection methods.
- Chapter 4: System Design and Implementation – Details the technical architecture, codebase structure, deployment strategies, and tools used.

- Chapter 5: Evaluation and Results – Presents the findings of empirical experments, including latency tests, cost analysis, and scalability scenarios.
- Chapter 6: Discussion – Interprets results in light of research questions and compares findings to existing literature. Summarizes the key contributions and outlines directions for future research.

Chapter 2: Literature Review

2.1 Introduction to Serverless Computing

The way developers create, implement, and maintain applications is being radically changed by the serverless computing paradigm, a major development in cloud-native software development. Teams are expected to oversee server provisioning, scaling, and patching in traditional models, which is a laborious and costly activity (Jonas et al., 2019b). With the elimination of infrastructure management, serverless computing relieves developers of this burden and frees them up to concentrate only on creating application logic. In most of the use scenarios, this method can significantly lower costs because it allows dynamic scaling, integrated fault tolerance, and a pay-per-execution payment model (Eismann et al., 2022)

Basically, the concepts indicates that developers are no longer in charge of managing servers, even though the name “serverless” may seem misleading because servers are still involved. Stateless, event-driven, ephemeral, and auto-scalable enviroments are used to carry out functions. With products like AWS Lambda, Azure Functions, and Google Cloud Funtions controlling the ecosystem, the Function-as-a-Service (FaaS) methodology embodies serverless thinking (Shafiei et al., 2022).

Apart from FaaS, another important component of the serverless environment is Backend-as-a-Service (BaaS) platforms such as Firebase, AWS Amplify, and Supabase. These platforms facilitate rapid development, particularly for front-end heavy applications like those built in React, by offering prebuilt features including cloud storage, real-time databases, authentication, and API administration (Jain, 2025). Serverless technologies are becoming more and more important in the discussion of web development because of their significance in full-stack development, especially when combined with contemporary JavaScript frameworks.

2.2 From Monoliths to Serverless: Evolution in Web Application Architecture

Web application architecture has evolved over time, moving from monolithic systems to those that are service-oriented and microservices-based, and ultimately to serverless architectures. Although they were easier to deploy at first, monolithic programs frequently proved challenging to manage as the codebase expanded. Vertical scaling, or increasing the power of a single machine, was the usual method of scaling monoliths, but this soon proved to be ineffective and expensive (“Microservices,” 2017).

By dividing applications into separate services that could be created, implemented, and scaled separately, microservices revolutionized the industry. This architectural style boosted flexibility but created new complexity in service orchestration, deployment, and monitoring (Newman, 2019). With their different capabilities for orchestration and containerization, Docker and Kubernetes emerged as the mainstays of microservices deployment. Yet, still they needed developers and DevOps teams to monitor resource usage, setup load balancers, and manage clusters, necessitating a high level of infrastructure expertise (Pahl et al., 2019).

Microservices logically led to serverless computing, which eliminated the need for developers to oversee even containers. Developers create distinct functions that are triggered by things like HTTP requests, database modifications, or planned tasks in a typical serverless context. Cloud platforms that automatically handle scaling, execution and fault tolerance host these functions (Baldini et al., 2017). This evolution has been referred to as the “natural trajectory” of web architecture, moving from tightly coupled monoliths to loosely coupled microservices and ultimately to stateless, on-demand functions (Van Eyk et al., 2018).

2.3 Serverless in the Context of React and Full-Stack Applications

The evolution of frontend development has changed significantly in parallel with the emergence of backend paradigms. Concerns are now more widely separated in application layers as a result of the growth of single-page applications (SPAs) and component based libraries like React. React, created by Facebook in 2013, revolutionized frontend programming by introducing a declarative UI model and virtual DOM (Ghosheh et al., 2006). The pairing of server-free on the backend and React on the frontend has gained popularity over time as an architectural structure that allows developers to create full-stack applications with little complexity. Refer Figure 3 to understand the growth.

Often called JAMstack (JavaScript, APIs, and Markup), this method consists of serverless functions managing backend functionality, React handling client-side rendering, and APIs facilitating communication between the two (Markovic et al., 2022) . The frontend of a JAMstack arrangement usually runs on static platforms like Netlify or Vercel, whereas serverless functions offer dynamic features like data manipulation, form processing, and authentication.

The concept is especially attractive because of its cost-effectiveness, scalability, and performance. While serverless functionalities automatically grow based on usage, static sites supplied by content-delivery systems (CDNs) load quickly. Additionally, development becomes simpler and maintainable when frontend and backend logic are separated. These characteristics fit in nicely with the requirements of contemporary small and startup companies, where limited resources and time-to-market are crucial considerations (Weyori and Tetteh, 2024).

2.4 Key Definitions and Scope

Function-as-a-Service (FaaS) models and Backend-as-a-Service (BaaS) solutions are both considered "serverless" for the purposes of this examination because they are both essential for the design of full-stack applications. The real-world implementation differs according on the platform and use scenario, but the fundamental ideas—scalability, lack of server management, and event-driven execution—remain the same.

Similarly, websites that use React on the front end and serverless services or APIs at the back end are often referred to as "full-stack React applications." Platforms like AWS Lambda, Google Firebase, or unique serverless frameworks made available by services like Netlify or Vercel can be used to create these.

By concentrating on these definitions, this literature study attempts to investigate the economic viability, developer experience, long-term maintainability, and technical feasibility of such architectures.

2.5 Cost Models and Scalability Trade-Offs in Serverless Computing

Serverless architecture is growing more and more popular in web development, particulary with full-stack tools like React, because to its cost-effectiveness and scalability. The “pay-as-you-go” concept of the serverless billing model makes it seem appealing, but the real cost behaviour is everything from simple. In an attempt to quantify serverless economics, a number of scholarly and commercial studies have discovered complex trade-offs between throughput, concurrency, provisioning models, and application design.

2.5.1 Serverless Billing Models: The Economics of Pay-per-Use

Compared to conventional Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) models, which charge users for reserved resources or uptime, serverless platforms bill per invocation, including the quantity of memory used and the period of time the function runs. One of the key advantages of serverless architecture, specifically for applications with low to medium traffic, is this pricing accuracy. As an example, AWS Lambda charges every 1 ms of execution, with a minimum charge length of 1 ms and a base memory allocation of 128 MB, according to Jonas et al. (2019a). Based on a comparison analysis by McGrath and Brenner (2017), apps running on AWS Lambda were up to 70% cheaper than their Amazon EC2-based competitors when used intermittently. For extended workloads, however, where containers or virtual machines (VMs) demonstrated more consistent cost and output efficiency, the savings dropped dramatically.

In addition, Wang et al. (2018) evaluated cold and warm start executions across Google Cloud Functions, AWS Lambda, and Azure Functions, showing that even though billing is still low in low-load scenarios, operational costs increase significantly due to increased memory allocation and execution time. In order to balance performance and cost, the analysis stressed the significance of optimal function tuning, which involves selecting the correct memory and timeout configurations. Serverless cost-efficiency depends on workload profile. Event-driven, unpredictable, or low-volume workloads benefit the most. However, high-throughput or consistently active services may experience cost inversion beyond a certain threshold (Hamza et al., 2023).

2.5.2 Scalability Advantages of Serverless Platforms

The promise of almost unending scalability is one of serverless computing's best selling elements. The cloud provider instantly handles distributing resources, concurrency limitations, and provisioning, and functions scale independently and horizontally. This feature can be especially useful in situations where user interactions are event-driven and traffic is unpredictable, such as frontend-heavy applications.
According to Eismann et al. (2022), serverless is "burst-scalable" by nature, which means it can manage hundreds to thousands of simultaneous executions without the need for human involvement. This feature enables a serverless backend that scales dynamically as the number of API calls increases to be connected to React-based frontends, which are frequently supplied statically via CDNs.

This stands in opposition to conventional container-based or virtual machine-based systems, which apply performance bottlenecks due to resource constraints and manual scaling setups. In fact, Ghorbian and Ghobaei (2025). used AWS Lambda for the backend and React for the frontend to create a real-time chat application that showed linear scalability under burst loads up to 10,000 concurrent users with steady response rates of 250 ms on average.

Scalability isn't always unrestricted, though. Regional concurrency constraints are enforced by the majority of cloud providers, which throttle operations if quotas are surpassed. For example, AWS Lambda has a 1,000 calls regional concurrency limit by default; this can be raised, but it will require administrative setup (AWS, 2024). Additionally, upstream and downstream services like databases, which may not scale as easily, may be impacted by associated executions. For example, unless a proxy layer or queuing mechanism is utilised, connecting thousands of concurrent Lambda computations to a single PostgreSQL database may lead to connection exhaustion (Khatri et al., 2020). Point to note here is, in a state of isolation serverless functions are quite scalable, but to ensure overall system scalability, they must be carefully coordinated with additional services like databases, caches, and APIs. Refer Figure 2 to understand it visually.

2.5.3 Cold Starts and the Scalability-Cost Nexus

One of the most frequently mentioned disadvantages of serverless platforms is cold start latency, which is the amount of time needed to initialise a new function instance in the absence of a warm container. Even though it doesn't directly influence price, it creates a performance hurdle that can have an indirect impact, particularly when developers use provided concurrency to get rid of cold starts.

Cold starts could take anywhere from 100 ms to more than 1.5 seconds, according to Wang et al. (2018) and Gao (2024), based on the runtime environment (Java vs. Node.js), package size, and memory allocation. AWS provides "Provisioned Concurrency," in which a specified amount of pre-warmed containers stay active to fulfil requests instantly, in order to prevent this. But this causes the pricing model to change to something more like reserved instances, charging customers for uptime even when the function is not used.

According to studies conducted by Leitner et al. (2019), cost efficiency with and without provided concurrency increased by 45–60% for functions with moderate load, even though it reduced latency. Developers must therefore carefully weigh the latency tolerance of their application against cost constraints. It's common practice to use a combination of approaches that keeps important endpoints warm while letting others grow as needed. Product display features in a React-based e-commerce platform, for example, might withstand cold starts, but provisioned concurrency is frequently used by payment processing and authentication endpoints to guarantee a smooth user experience (Blessing, 2025; Ebrahimi et al., 2024). Its important to note that, although cold starts aren't necessarily expensive they still bring important performance decisions that could lead to developers overprovisioning, which would reduce overall cost effectiveness.

2.5.4 Comparing Serverless to Containers: A TCO Perspective

In a comprehensive review of serverless and container-based deployment experiments, Syeda et al. (2025) discovered that for workloads with irregular or spiky traffic, serverless systems (like AWS Lambda) frequently lower TCO by about 40–60% over a 30-day timeframe. On the other hand, when traffic patterns are steady and without interruption, containerised deployments—like those powered by Kubernetes—usually provide more accurate cost models and greater compatibility with monitoring systems. Furthermore, serverless deployments usually allow for quicker launch cycles and less developer overhead; yet, for complex applications, Kubernetes-based systems might offer more reliable debugging and long-term maintainability.

Similar to this, Fan et al. (2020) created a cost model using simulated user traffic and showed that a well-optimized container system performed far better than serverless in terms of both cost and latency in high-volume SaaS applications. However, serverless was greater in both areas for internal tools, MVPs, and seasonal workloads. A notable point is that, Although

serverless computing. technologies excel in speed and scalability, they may not always be more affordable. In addition to computing expenses, TCO calculations need to account for human resources, tools, and operational overhead.

2.5.5 Pricing Pitfalls and Hidden Costs

Despite the theoretical uniformity of serverless pricing, a number of studies have shown indirect or hidden expenses that may impact total cost-effectiveness. These include the following:

- **Vendor Lock-In**: Since serverless applications are intrinsically linked to the proprietary function syntax and configuration rules of their hosting providers, migrating between platforms presents significant obstacles and long-term switching costs. (Zhao et al., 2022).
- **Third-Party API Costs**: External services like authentication, storage, and database solutions can be responsible for the majority of costs for serverless workflows on AWS, frequently making up as much as 83% of total expenses. If not properly managed, the combined effect of services like Cognito, DynamoDB, and S3 billing individually could result in surprisingly expensive cloud bills. (Marcelino et al., 2025).
- **Logging and Monitoring**: Along with many serverless apps, AWS CloudWatch charges per gigabyte for log ingestion and storage, and logging fees can quickly take over the bill at big volumes. Although AWS currently provides various delivery routes and tiered pricing (e.g., $0.50 to $0.05 per GB) to cut costs, high-volume systems may experience logging costs that are equal to or higher than compute costs in the lack of filtering, retention management, or batching (Pandey and Barker, 2025).

Additionally, although Firebase and similar platforms are useful for developing prototypes, their price tiers increase substantially as usage beyond free or hobby criteria. Jain et al. (2025) discovered that reads/writes on Firestore alone caused an eight times rise in monthly expenditures for their React-Firebase application following an unexpected spike in user traffic. Serverless pricing can be surprisingly straightforward. Logging, database utilisation, third-party services, and possible migration limitations must all be taken into consideration in a true cost analysis.

Although serverless architecture provides a compelling cost and scalability value proposition, its actual economical behaviour is complex. Important elements that affect the overall cost-

effectiveness of serverless installations include provisioned concurrency, workload fluctuation, cold start mitigation techniques, and third-party service utilisation. When properly constructed, this architecture can be extremely useful for React-based full-stack apps. Nevertheless, it must be avoided to avoid common problems such as inappropriate use of supplied concurrency or database saturation at scale.

2.6 Performance Analysis of Serverless Systems – Cold Starts, Latency, and Concurrency

The performance characteristics of the serverless architecture are important to its success in production settings, especially with full-stack apps that use frontend frameworks like React. A serverless system's ability to sustain satisfactory performance under real-world workloads depends extensively on its execution overhead, cold start latency, and concurrent scalability. This section offers an in-depth investigation of these important performance indicators and synthesises results from current studies conducted by educational testbeds and cloud providers.

2.6.1 Understanding the Cold Starts in Serverless

When a cloud service provider must initialise a new container instance to process a newly received request since there isn't an idle instance available, this is known as a "cold start." This bootstrapping procedure involves allocating resources, loading runtime environments, and executing initialization code. A warm start, on the opposite hand, executes much more quickly since it makes advantage of an already-running container.

One of the first groups that measured cold start latency consistently across well-known serverless platforms was Wang et al. (2018). According to their conclusions, cold start latency differed greatly between runtime environments and suppliers. For example, some Java functions could take more than 1,200 ms to start, while Node.js functions from AWS Lambda showed cold starts in the 200–500 ms range. The allocation of memories was also shown to be important in their trials; larger memory (e.g., 1024MB vs. 128MB) generally resulted in shorter cold start times since it dedicated more computing resources.

For serverless platforms, Lee et al. (2024) suggest SPES, a prediction-driven scheduling system that separates invocation patterns and pre-warms functions appropriately. Although their tests show that cold starts (such as 75th-percentile delay) can be reduced by over 50%, the method depends on an external scheduling component that might be unneeded for smaller-scale applications. With no free tier and ongoing charging even during idle times,

provisioned concurrency comes with ongoing fees determined by the chosen memory size and concurrency level. For example, it might cost about $10 per instance per month to maintain a 1 GB Lambda with supplied concurrency. Therefore, considering traffic patterns, enabling provided concurrency across three crucial tasks could raise monthly expenditures by 30 to 50%. Because of this cost profile, numerous developers choose to leave several endpoints in default (on-demand) mode and save the feature for latency-critical endpoints like login, processing payments, or dynamic forms (AWS, 2025a). Performance unpredictability is introduced by cold starts, particularly for first-user encounters or occasional queries. To prevent customer experience degradation, provided concurrency and warmup techniques must be used intentionally.

2.6.2 Latency Benchmarks in Full-Stack Serverless Applications

One important element of performance is latency, particularly in programs that interact with users. It addresses the time needed for server-side processing, the time it takes for frontend requests to go to the backend, and the time needed to reply to the client.

According to the standards of the industry and AWS data, warm invocations, particularly with lightweight runtimes like Node.js, usually stay under 300 milliseconds, whereas AWS Lambda cold starts can add latency of more than one second, occasionally up to two or more. This strategy is frequently seen in serverless API stacks that are supported by API Gateway plus Lambda and fronted by SPAs hosted on CloudFront (J. Beswick, 2021).

In terms of cold start latency, industry benchmarks regularly indicate that AWS Lambda runtimes, such as Node.js and Python, perform noticeably better than heavier runtimes, like Java and C#. In comparison to Python or Node.js, Java and.NET had cold start times that were more than 100× slower in supervised tests, which resulted in slower overall responsiveness. For developers to achieve low-latency behaviour in serverless deployments, the programming language selection is fundamental (Cui, 2017).

According to AWS guidelines and community assessment, connection pool exhaustion could lead serverless Lambda functions accessing relational databases under high concurrency to run less efficiently. Until the database's maximum connection threshold is reached, each concurrent Lambda instance frequently keeps up its own database connection. Request failures and increased query latency result from this. For high-scale use cases, AWS highly

recommends switching to scalable alternatives like DynamoDB or Redis, or adopting RDS Proxy to efficiently pool and multiplex database connections (Lapin, 2021).

Some mainstream platforms have added edge functions to reduce latency. By conducting logic near the user and using Cloudflare Workers, Vercel's edge functions minimises geographical latency. These are especially advantageous for React-based global applications where features like A/B testing and authentication must operate rapidly and internationally. Although it differs depending on the runtime environment, backend service configuration, and function cold state, latency in serverless systems is often acceptable for use cases. Perceived latency can be decreased even more using edge computing and the best language selection.

2.6.3 Concurrency and Load Behaviour in Serverless Platforms

A key performance characteristic for assessing how serverless architectures scale in practical circumstances is concurrency, or the capacity to manage multiple concurrent invocations. Serverless technologies typically abstract these decisions, in contrast to traditional server-based systems that need human configuration for handling concurrent requests.

This is known as elastic concurrency, according to Eismann et al. (2020), in which the platform scales function instances automatically in accordance with demand. Function invocations in their AWS Lambda studies raised more linearly with incoming HTTP requests up to the soft concurrency limit set by the service provider (by default, 1,000 concurrent executions per region). Unless a quota increase had been requested, throttling started after this.

A major conclusion from their research is that supporting services need to expand in order with serverless functions. Connection bottlenecks will arise, for example, if 1,000 Lambda functions access a single RDS (Relational Database Service) instance at once. Because of this, high-concurrency systems are frequently encouraged to use serverless-native databases like DynamoDB or Aurora Serverless.

When auto-scaling is working at its best, high-throughput AWS Lambda + DynamoDB backends can manage thousands of concurrent requests (e.g., 10,000 RPS with appropriate concurrency configuration) with minimal latency. However, without configured autoscaling policies, performance degradation may occur when DynamoDB throughput limits are reached, requiring manual capacity adjustments. This emphasises how essential it is to set up scalability appropriately for serverless e-commerce operations that are subject to latency (AWS, 2025b; Bandara, 2024; Prasad, 2021).

While Google Cloud Functions' standard approach enables each instance to manage many simultaneous HTTP requests (up to about 80), Azure Functions offers adjustable per-instance concurrency, such as multiple processes or message bindings per function host. AWS Lambda, on the other hand, maintains strictly to the one-invocation-per-instance principle. These concurrency models have an immense effect on productivity and memory efficiency. For highly concurrent I/O-bound applications, AWS Lambda needs more containers (and memory) to grow, while Azure and Google can handle more requests on lesser instances (AWS Lambda, 2025a; Microsoft Learn, 2025). The supporting infrastructure must be created to handle serverless solutions' ability to grow under concurrent load. End-to-end performance is influenced by resource quotas, API rate constraints, and database selection.

2.6.4 Memory Allocation and Performance Tuning

Since majority of platforms distribute CPU resources proportionately, memory allocation in serverless functions impacts both available RAM and CPU resources. For jobs involving computing, managing huge files, or network operations, this has a consequence on performance.

As reported by McGrath et al. (2017), boosting memory in AWS Lambda from 128MB to 1024MB minimised cold start latency and shortened the overall execution time for CPU-intensive functions in fifty percent. But the price also went up proportionately, so tuning is now more of an optimisation exercise than a simple update.

Adaptive resource allocation techniques that dynamically modify memory and compute units based on earlier invocations were proposed by Shillaker and Pietzuch (2020). Despite being theoretical, their calculations indicated that they could save 20–30% on the total expenditure of the function while maintaining an acceptable latency.

Practically speaking, programmers are encouraged to use actual traffic statistics to evaluate routines and modify memory allocation as required. Higher RAM settings can frequently be advantageous for React-based apps that use serverless for data APIs, media uploads, or analytics in order to guarantee faster performance. Allocating memory and CPU has a big impact on function performance and cost. Optimisation of serverless performance require adaptive tuning that is informed by actual usage patterns and benchmarks.

2.6.5 Language Runtime and Dependency Overhead

A serverless function's the time of execution selection affects execution performance as well as cold start timings. Because of their quick start-up times and simplicity of integration with frontend frameworks like React, Node.js and Python are generally chosen.

Node.js and Python may start considerably quicker and with greater effectiveness than Java and.NET, which have far higher latency and require more memory for satisfactory performance, according to Aleksandr Filichkin's (2021) benchmark of AWS Lambda runtimes. Emin Bilgic's industrial research further confirms that Node.js and Python maintain nearly comparable performance across cold and warm calls, while Java's cold start duration could go beyond several seconds. These findings point out that Node.js is still an extremely popular choice for full-stack JavaScript applications, especially for serverless situations where latency is a concern (Bilgic, 2022; Filichkin, 2021).

Bloated Lambda deployment packages that included pointless libraries, like an unoptimised React-based CMS, have been found by industry practitioners to cause cold starts that take longer than 1.5 to 2 seconds. Cold-start latency often drops to well below 600 ms with thorough pruning, tree-shaking, and the use of modular or lighter SDKs (such as the AWS SDK v3), highlighting the crucial role that package optimisation plays in serverless performance (Aslam, 2025; Beswick, 2021; Gomes, 2021).

Developers can create smaller serverless bundles by including modern tools like Webpack, Rollup, and esbuild into CI/CD pipelines. Serverless performance is significantly affected by bundle size and language selection. Smoother user experiences and quicker cold starts are the consequences of lean, optimised deployments.

Serverless computing effectiveness is still a double-edged sword. Cold starts, dependency overhead, and improperly configured infrastructure may negatively impact the experience, even while warm-start latencies, concurrent scaling, and elasticity are advantages. Performance-aware deployment techniques, including as memory optimisation, runtime selection, and backend service orchestration, can significantly benefit React full-stack apps.

2.7 Design Patterns and Integration Models for Full-Stack React and Serverless Architectures

The method in which developers organise and implement contemporary web apps has evolved due to serverless architecture. Serverless allows for the quick development of full-stack applications with increased scalability, modularity, and operational efficiency when paired with component-driven frontend frameworks like React. But the lack of a server is not an indication that there is no architecture. Actually, significant architectural preparation is required for serverless deployments to be successful, especially when creating complicated, stateful, or high-availability full-stack systems. This section examines best practices found in recent studies and enterprise deployments, as well as commonly used serverless design patterns and React-backend integration models.

2.7.1 Backend-for-frontend (BFF) Pattern

One popular architectural paradigm for full-stack React apps is the Backend-for-Frontend (BFF) pattern. A specialised serverless backend is developed under this paradigm to meet the requirements of a particular frontend or user interface layer (Richardson, 2018). Improved optimisation and API response customisation are made possible by this separation of the frontend from backend complexity and domain-specific business logic.

Based on their capacity to balance coherence and client-specific flexibility, Backend-for-Frontend (BFF) patterns are frequently used in serverless microservice architectures, according to a practitioner study (Oliveira, 2020; Waseem et al., 2021). Because component interactions and user interface state are closely related in React apps, a BFF layer helps maintain clear data boundaries and steer clear of frequently occurring issues like over- or under-fetching from general-purpose API levels.

Firebase Cloud Functions are frequently utilised by community practitioners as a BFF layer for React dashboards, particularly to format Firestore data to minimise client-side over-fetching. BFF endpoints centralise access patterns in these configurations, streamlining the frontend and cutting down on pointless data retrieval. When accessing functions straight from the user interface, developers report cold start latency issues (often >1 s) and stress the value of caching or establishing minimum instance policies to control performance (Reddit, 2024a, 2024b). The BFF pattern improves response time and maintainability in React-serverless stacks by permitting customised backend logic for every client application.

2.7.2 Event-Driven Design and Function Granularity

Serverless computing is fundamentally event-driven. Reactive, decoupled architectures are encouraged by the way triggers, including HTTP requests, database events, file uploads, or message queue updates, call functions. Micro-function granularity had been suggested by (Hendrickson et al., 2016), who recommended that each function should be accountable for a single business task. Although this kind of deconstruction is good in theory, it can result in an explosion of deployable units, which would make operations more complex.

In serverless systems, adaptive granularity strategies—like those put out by Hui et al. (2025)—emphasize the trade-offs between coarse- and fine-grained functions. In accordance to their findings, finely decomposed functions are more effective for rare, loosely connected tasks, while coarse-grained function decomposition proves more appropriate for high-performance paths or behaviours involving cross-cutting concerns (like authentication or logging). Architectural decisions for coherence, latency, and scalability can be guided by this granularity decision model (Hui et al., 2025).

Developers frequently pick a hybrid approach while building React applications. Data-intensive endpoints (such as profile sync) and authentication flows (like login and token refresh) are usually built as once-only operations. In the meantime, related activities (such creating, updating, and deleting users) may be bundled into coarser-grained Lambda handlers employing administrative interfaces.

According to an analysis of 89 serverless apps of production quality, the majority of them had fewer than ten functions—82% had five or lesser, and only approximately 7 percent had more than ten. This low granularity suggests that closely connected business logic frequently limits function reusability between routes. These results highlight the architectural advantage of using a Backend-for-Frontend layer to prevent over-fetching or under-fetching in React-driven applications and provide reusable, client-specific data boundaries (Eismann et al., 2020). Function granularity needs to be in line with the performance requirements and complexity of the application domain. Integrating functions with UI components frequently ends up in a logical and manageable framework for React-based user interfaces.

2.7.3 API Gateway Integration and Routing Strategies

API Gateway services serve as connectors between frontend clients and backend logic in full-stack React + serverless apps. They offer request validation, throttling, authentication, and HTTP routing. Two of the most frequently used services for this are Firebase HTTPS triggers and AWS API Gateway.

In an effort to streamline routing and minimise mapping overhead, industry practitioners have suggested classifying serverless RESTful endpoints under logical base pathways (such as /api/users, /api/posts) and utilising API Gateway's Lambda proxy interface. Proxy integration produces more constant latency under burst traffic than non-proxy HTTP integrations, as demonstrated by Josh Durbin's performance proof of concept. Additionally, based on architecture guidelines by Jeremy Daly and others, microservice configurations can be made simpler, more cohesive, and less prone to data over-fetching by combining related endpoints and leveraging proxy integration (Durbin, 2017; Hefnawy, 2016).

The authors of the 10th International Serverless Computing Workshop, "GraphQL vs. REST: A Performance and Cost Investigation for Serverless Applications," compares serverless GraphQL (with AWS AppSync) to REST APIs for a variety of workloads. They noticed that GraphQL continuously improves round-trip latency and data-fetching efficiency, two performance advantages that are especially noticeable in bursty or high-latency situations and that directly result in speedier frontend rendering in React-based applications (Jin et al., 2024).

But GraphQL brings complexities of its own, especially in serverless scenarios. In large applications, resolver functions can become extensively layered and challenging to narrow down. Some of this complexity is reduced by tools like Apollo Server and Dgraph, because they offer performance monitoring and prebuilt resolvers. React frontends and serverless backends are capable of being seamlessly integrated with API Gateway and GraphQL. While GraphQL provides performance improvements in data-intensive user interfaces, REST is still easier to scale and debug.

2.7.4 Secure Authentication and Session Management

Identity providers (IdPs) such as Firebase Auth, AWS Cognito, or Auth0 are frequently used to manage authentication in serverless apps. React apps assign the frontend, which communicates with serverless functions that carry out authorisation, authentication requirements (such as sign-in as well as session token management).

The challenge of over-permissioned serverless IAM settings has recently been addressed by recent studies and open-source technologies. According to Marin et al. (2022a), the absence of dynamic enforcement mechanisms in serverless systems causes static IAM roles to frequently give excessive rights. By default, tools such as AWS Access Analyser and IAM Zero aim to decrease the attack surface by automatically scoping permissions according to real runtime behaviour. These strategies are especially crucial for admin-like apps where multiple scenarios of use (such as read-only versus write flows) call for clearly defined access (Chintala et al., 2018; “Implementing IAM in Serverless Architectures,” 2025; Marin et al., 2022a).

At the API Gateway layer, token verification and role-based access control (RBAC) are important security enforcement approaches. Before executing backend requests, a React frontend might, for example, store an access token in localStorage while having an authoriser function authenticate it.

Replay attacks and XSS, or cross-site scripting, are two of the major security vulnerabilities that can result from online applications' improper handling of authentication tokens, according to OWASP. OWASP advises utilising HttpOnly and Secure cookies for refresh tokens and storing access tokens in memory in React-based and serverless systems. By restricting tokens from being immediately accessible through JavaScript, this lowers the possibility of XSS exploitation and preserves session continuity by employing safe token rotation techniques (OWASP, 2025). Identity providers, backend IAM policies, and React frontend logic must work together closely to guarantee secure authentication in serverless applications. Secure token techniques and least-privilege design are necessary.

2.7.5 CI/CD and Infrastructure-as-Code in Serverless Projects

Although runtime infrastructure is abstracted by serverless platforms, structured tooling is still essential to the deployment of consistent and dependable backends. AWS SAM, Terraform, and Serverless Framework are some of the tools that are frequently utilised in Continuous Integration and Continuous Deployment (CI/CD) processes for serverless application development.

According to both practitioner surveys and community sources, connecting GitHub Actions with the Serverless Framework significantly improves deployment automation and consistency for React + serverless applications. Automated pipelines certainly enhance release cadence

and consistency, even while precise setup-time improvements (such as 40%) are not recorded. Nevertheless, user feedback and survey findings show that configuring automation workflows (including YAML and CI pipelines) is still an essential pain point, with debugging and error diagnostics being frequently mentioned as obstacles (Amareswer, 2025; Saroar and Nayebi, 2023).

Developers can declaratively design functions, APIs, databases, and permissions using Infrastructure-as-Code (IaC). Serverless resources can be version-controlled and replicated between environments with the help of solutions like Terraform or AWS SAM. For example, a single YAML or HCL configuration file can be used by a developer to spin up the staging, production, and test environments for a React + Lambda applications.

IaC decreases the chance of configuration drift and promotes transparency in multi-developer teams. IaC templates, however, can become verbose and increasingly difficult to manage as programs get more complex, particularly when handling multiple permissions and secrets. Serverless deployments benefitted greatly from the framework that CI/CD and IaC tools provides. Full-stack application development has been simplified by its integration with Git-based workflows; yet, in the absence of modular design, complexity can grow dramatically.

It takes more than merely a handful of cloud services for developing full-stack applications with React and serverless backends that are secure, maintainable, and fast. Scalable systems can be designed using patterns like BFF, event-driven granularity, GraphQL, and safe IAM enforcement. They provide a strong development lifecycle for contemporary serverless programs when utilised alongside IaC and CI/CD pipelines.

2.8 Tooling Ecosystem and Monitoring for Full-Stack Serverless React Applications

Serverless architecture is not without complexity, even though it gives developers an abstraction from infrastructure administration. Deployment pipelines, observability, configuration drift, and real-time debugging continues to be challenges for developers, particularly when it comes to full-stack apps with serverless services on the backend and React on the frontend. In addition to platforms like Vercel, Netlify, and AWS Amplify, this section covers the essential elements of the serverless tooling ecosystem, including logging, monitoring, CI/CD, and infrastructure-as-code (IaC) technologies that facilitate professional-grade deployments.

2.8.1 Developers Platforms: Vercel, Netlify, and AWS Amplify

Full-stack development has been reinvented by contemporary systems such as Vercel and Netlify, which combine frontend and backend operations into a single deployment pipeline. The above options, that involve preview deployments, integrating serverless functionality, quick static site hosting, and Git-based version control, are highly compatible with React.

Platforms like Netlify significantly speed up end-to-end deployment for React SPAs combined with serverless backends, according to industry documentation and developer comments. These products enable functional deployments (including React + Lambda) in just under an hour through providing ready-to-use templates, integrated continuous deployment, and automatic DNS management (Netlify, 2025a).

In particular, Vercel further optimises with native support for edge functions, global distribution, and hybrid rendering (e.g., SSG + SSR via Next.js). Dynamic logic occupies space near end users on its edge network, delivering TTFB, cold start, and response time improvements that are more challenging to accomplish through traditional RESTful or Lambda-based methods (Wachtel, 2022).

Amplify by Amazon Web Services has a more powerful enterprise-aligned feature set, according to industry comparisons and feedback from the community. This is especially true given its close connection to authentication, AppSync (GraphQL), and DynamoDB, which can be highly beneficial in complex student portal or education IT scenarios. But, in order to be maintained efficiently, Amplify usually needs a more lengthy ramp-up and more complicated configuration than Firebase and Netlify (Mitchell, 2024; Savants, 2025; Ulili, 2025).

Although vendors like Vercel and Netlify streamline front-end development and deployment, they depend primarily upon proprietary conventions, such as edge-function syntax and routing models, resulting in portability difficult. Likewise, AWS Amplify closely connects apps to AWS services like Cognito, AppSync, and Lambda, even if it abstracts away some infrastructure complexity. The negative impact of moving away from these platforms is a concealed switching cost (Asplund, 2022; Satzger et al., 2013).

Serverless vendor lock-in applies across compute and storage categories, as Zhao et al. (2022) demonstrate, the necessitating multi-cloud-aware architectures for allowing for genuine mobility. Data coupling and proprietary APIs are commonly mentioned by industry

professionals as the two major obstacles to vendor flexibility (Cui, 2020). While AWS Amplify provides deeper backend communication at the expense of complexity as well as potential lock-in, Vercel and Netlify allow for quick deployment and flawless development for React-based serverless apps.

2.8.2 Continuous Integration and Deployment Pipelines

When multiple environments (development, staging, as well as production) need to be operated consistently, serverless development gets significant benefit from well-integrated CI/CD pipelines. When trying to automate testing and deployment, Serverless Framework, Terraform, or AWS SAM are frequently used in conjunction with tools like GitHub Actions, GitLab CI/CD, and Bitbucket Pipelines.

Information from the community and industry demonstrates that serverless deployments are much accelerated by CI/CD automation. Compared to manual CLI-based deployments, surveys show that delivery speeds have boosted by up to 77%, and many teams estimate that processes have improved by at least 50%. When working with React, developers frequently automate builds (for example, npm run build → publish to S3 or Netlify) after which they use the Serverless Framework in CI/CD to deliver serverless backends. Troubleshooting pipeline setups (such as YAML problems and step failures) is still an important problem, despite how this increases release velocity and consistency (Helendi, 2019; Zetas, 2025).

The infrastructure promotion, in which IaC templates include resources and environment-specific adaptations (such as database URIs and authentication tokens), is a fundamental approach. Human error can be reduced via automation, which guarantees that these configurations are accessible and consistent.

Dependency on manual CLI operations is significantly reduced by CI/CD Automation techniques like AWS CodePipeline and branch-based deployment triggers (e.g., staging from develop, production from main). The automation of the release process reduces deployment failures and rollback events, as reported by AWS, while industry customer case studies demonstrate a 95% reduction in deployment faults. Even though formal comparisons of percentage reductions (e.g., 35%) are not recorded in academic literature, these efficiencies are consistent with increased production stability and decreased incident rates (Andersen, 2025; Cloudthat, 2025). Implementing CI/CD to automate serverless deployments increases

dependability, accelerates release cycles, and facilitates accurate multi-environment management of configurations.

2.8.3 Infrastructure-as-Code (IaC) and Environment Management

Serverless services provides an abstraction, but provisioning, managing, and version-controlling infrastructures is still necessary. With Infrastructure-as-Code (IaC) tools that include AWS CloudFormation, Terraform, Pulumi, and Serverless Framework, developers can employ declarative templates to define backend infrastructure.

In serverless architectures, practitioners suggest treating infrastructure as a first-class citizen, especially when a stack has many S3 buckets, Lambda functions, and DynamoDB tables. Deployment across environments is consistent and version-controlled when standardised IaC templates are used (via CDK, SAM, Terraform, or Serverless Framework). According to AWS professionals, this facilitates scalable, productized deployments in serverless apps, controls configuration sprawl, and minimises human error (Beswick, 2024, 2020; Marquez, 2023).

Over 76% of respondents, many of whom came from startups, said they preferred the Serverless Framework as their primary instrument for building and deploying serverless architectures, according to Serverless, Inc.'s 2025 "State of the Serverless Community" poll. In a single serverless.yml file, functions, environment variables, permissions, and event triggers are provided (Gottlieb, 2016).

Teams who currently utilise Terraform for expanded infrastructure provisioning acknowledge it because it offers excellent multi-cloud support despite its high level of complexity. Terraform, for instance, is frequently used by organisations that deploy both serverless APIs and Kubernetes clusters for consistency. The infrastructure as Code decreases environment drift and misconfiguration while enhancing teamwork, security, and repeatability in applications that are serverless.

2.8.4 Observability and Monitoring: The Invisible Complexity

In serverless systems, monitoring as well as observability continuing to be important challenges. Conventional logging and tracing techniques sometimes fail because functions are dispersed, stateless, and have short lifespans. In full-stack applications, this becomes especially problematic because a user action in the React frontend may initiate several backend processes, each of which may produce logs individually.

Shillaker and Pietzuch (2020) draw emphasis on the disadvantages of serverless platforms' ephemeral, stateless isolation and recommend Faaslets, which are lightweight WebAssembly-based containers that facilitate memory sharing and more effective function-level isolation, enhancing debuggability and traceability in high-function-count settings. In addition, Borges et al. (2021) describe a serverless troubleshooting architecture that combines OpenTelemetry with AWS X-Ray. Distributed tracing, as demonstrated by their prototypes, significantly improves visibility across intricate serverless compositions, which is crucial for recognising errors and performance issues within multi-function systems.

In the opinion of Borges et al. (2021), using OpenTelemetry and AWS X-Ray to instrument AWS Lambda applications significantly boosts fault localisation in serverless operations, strengthening observability and reducing debugging effort. Other industry sources demonstrate how integrating metrics, logs, and traces across application stacks can speed up mean time to resolution (MTTR) in production circumstances, despite the fact that they neither use frontend browser tracing nor measure the reduction in MTTR (Leffler, 2021).

Because AWS CloudWatch, Azure Monitor, and GCP Stackdriver utilize various interfaces, query languages, and pricing structures, it can be hard to accomplish consistent observability using native cloud tools. This is highlighted in practitioner literature and developer comments. Because third-party observability solutions like Datadog or New Relic offer cross-cloud dashboards, unified alerts, and streamlined tracing, many teams prefer to use them regardless the fact that they are more expensive and require more setup effort. (Serverless Savants, 2025). In serverless applications, observability requires to be specifically planned for. To decrease debugging friction, distributed tracing, centralised logging, and frontend-backend correlation are necessary.

2.8.5 Debugging and Local Development Tools

Lack of local compatibility with cloud settings is one of the most commonly mentioned developer pain points in a serverless development. Because of platform-specific behaviours, IAM configurations, and cloud services, serverless tasks are more challenging to replicate precisely on a local computer than typical server or container-based apps.

Developers frequently use local emulation tools like serverless-offline, SAM CLI, or Docker-based stacks for replicating serverless environments locally, as described by Chen et al. (2021) and supported by AWS prescriptive guidance. Key behaviours, such cold start latency,

IAM policy enforcement, or availability features of services like DynamoDB and S3, cannot be properly reproduced by these technologies. Because of this, React + serverless developers usually use mock backend services for frontend testing, which lowers test fidelity and complicates maintenance. Additionally, empirical results show that developers still encounter differences from production environments even when they use integrated local emulation technologies (e.g., Serverless Framework, LocalStack) (AWS, 2025c).

The Firebase Emulator Suite is crucial for quickly iterating on Firestore security rules, authentication flows, and Cloud Functions—without raising cloud usage fees or compromising production datasets, according to developer experiences. Developers can develop, test, and validate code and infrastructure locally thanks to LocalStack's ability to fully emulate AWS environments offline, including Lambda, API Gateway, S3, and DynamoDB services. These techniques improve productivity while significantly reducing risk and development costs (AWS, 2025d; Boa, 2025)

Nevertheless, complete balance is rarely attained. Feedback loops become slowed down by the need to deploy into actual cloud systems in order to debug problems like IAM permissions or edge-case latency. Although it is not yet smooth, local development in serverless projects continues to get better. Although they reduced iteration costs, emulators and cloud sandboxing cannot be as effective as full fidelity testing.

In conclusion, the foundation of a scalable and maintainable serverless React application is made up of developer platforms and infrastructure. Each level of the stack needs dedicated focus, from real-time observability with OpenTelemetry to deployment automation with GitHub Actions. While platforms such as Vercel, Netlify, and AWS Amplify streamline development, they also add complexity brought forth by abstraction. The most effective full-stack serverless teams use distributed monitoring, CI/CD automation, and Infrastructure-as-Code as fundamental approaches to development.

2.9 Security, Compliance, and Isolation in Serverless Architectures

Serverless computing raises major safety challenges, particularly for implementing multi-user, data-sensitive applications, in spite of its benefits in scalability and abstraction. The cloud provider and the developer share responsibility for security in full-stack React apps which use serverless APIs. The security models, runtime boundaries, tenant isolation, and compliance limitations related to serverless platforms are examined in this section.

2.9.1 Shared Responsibility Model in Serverless

With serverless platforms including AWS Lambda, Google Cloud Functions, and Azure Functions, the developer is in charge of the function code, configuration, and data-level security, whereas the provider safeguards the infrastructure (such as virtual machines, storage, and networks). However, this concept frequently becomes harder to comprehend by the temporary and fine-grained nature of serverless services. In order to prevent disclosure between invocations, developers must design stateless functions, avoid overly strong roles, and expressly enforce identity segregation (Li et al., 2021; Marin et al., 2022a).

Critical concerns related to excessively permissive IAM roles in serverless applications have been highlighted by a number of studies. For example, using wildcard permissions (e.g., "Action": "*") and extensive resource access, particularly with Lambda functions and S3 resources, are common fault lines that lead to privilege escalation and data breaches, as stated in AWS's own best-practice guidance and security analysis reports (Magee, 2025). Real-world consequences are further demonstrated by a CyberSapiens case study: Overly broad access was possible due to excessive IAM permissions throughout a SaaS provider's implementation, highlighting the necessity for establishing stricter "least privilege" regulations (CyberSapiens, 2025).

This trend has been backed by empirical research in the context of Infrastructure as Code (IaC). While access control policies are frequently established, encryption and safe secret handling are frequently overlooked, according to a study looking at 812 open-source IaC projects on GitHub. This suggests that practitioners frequently do not have sufficient expertise about secure setup (Verdet et al., 2025).

All of these results indicate that developers are frequently setting up serverless IAM roles incorrectly, allowing them more access than they really need. A methodical approach is necessary for tackling this problem: implement automated tooling or code-review procedures that identify over-privilege, conduct frequent audits, and enforce resource-scoped, fine-grained policies—especially in serverless systems that are functionally rich and changing quickly.

2.9.2 Authentication and Authorization in Full-Stack Serverless Applications

Identity providers like AWS Cognito, Firebase Authentication, Auth0, or Okta typically remain in charge of authentication in serverless configurations. After authentication by the user,

access tokens (JWTs) are delivered to API endpoints for validation using React-based frontends.

Because token-based authentication use self-contained credentials, it is naturally suitable with stateless serverless operations. However, when this approach is implemented insecurely, major challenges are introduced. Developers ought to accidentally save tokens in dangerous places (like localStorage) or overlook to verify critical assertions like issuer (iss) and expiration (exp). To maintain security across dynamically scaled, transient serverless settings, research on token-based authentication in cloud-based microservices systems highlights the significance of rigorously enforcing token validity and carefully monitoring token use (Venčkauskas et al., 2023). Additionally, broader examinations of serverless security problems show how the stateless structure of functions and fractured execution boundaries make it more difficult to secure authentication flows (Li et al., 2021; Marin et al., 2022b).

Custom authoriser techniques, which check incoming tokens before to forwarding requests to the backend, are supported by AWS API Gateway in order to prevent this. For comparable enforcement at the database and storage layer, Firebase utilises security rules. When correctly implemented, these techniques guarantee consistent authorisation and lessen the psychological strain on backend engineers.

By extending AWS Cognito with user pool groups and custom JWT claims (such as cognito:groups) to gate access to serverless activities, numerous applications implement role-based access control. This approach is particularly beneficial in educational environments where different roles—students, teachers, and administrators—have different rights. By directly incorporating group identifiers into tokens that get issued upon authentication, Cognito makes role-based authorisation easier. While being scalable to single-page real-world applications, developers frequently point to the absence of a centralised policy engine as a drawback, which makes it more challenging to manage and develop large-scale RBAC deployments (Amazon Congnito, n.d.; Leiß, 2025).

2.9.3 Data Protection, Secrets Management, and Secure Function Execution

The management of secrets, including database URIs, credentials, and API keys, is an important aspect of function-level security. Instead of being hardcoded, these need to be securely saved and injected at runtime.

Since environment variables are easy to use and practical, they are regularly used to inject configuration data in serverless applications. But there are some security hazards associated with this practice. Even when AWS Lambda encrypts environment variables while they are not in use, anyone with function configuration access continues to see them, making sensitive data like database credentials or API keys unintentionally vulnerable to attack (AWS Blogs, 2022; AWS Lambda, 2025b). According to study by Orca Security, more than 26% of Lambda functions reveal secrets through environment variables, an issue frequently caused by developers who put usefulness ahead of secure management (Lewis, 2021).

Many experts recommend using dedicated secrets-management systems like AWS Secrets Manager, Azure Key Vault, or Google Secret Manager due to of these dangers. Without the vulnerabilities brought on by environment variables, these services provide centralised examinations, automatic rotation, fine-grained access control, and safe storage (DEV Community, 2025; Douglas, 2019). The AWS Compute Blog, for example, specifically cautions against keeping secrets in Lambda environment variables and recommends utilising Secrets Manager to safely retrieve them during runtime (AWS Blogs, 2022).

Security throughout runtime is also crucial. Sandboxing functions is a good way to prevent escape attacks. In order to decrease cold start penalties while preserving runtime isolation, Shillaker and Pietzuch (2020) examined at the Faasm runtime and suggested lightweight containerisation. By restricting functions from sharing memory or filesystem states, their design limited the possibility of privilege escalation and data leakage.

If IAM roles are not properly scoped, function chaining—invoking another serverless function—introduces serious safety risks. Attackers can increase privileges laterally throughout the system by taking full advantage of the excessive trust that exists between chained functions. These concerns have been backed by a comprehensive study that was published in the Journal of Cloud Computing. The authors emphasise that lateral privilege escalation can result from incorrectly setup roles or unscoped trust relationships, and that functions generally hold additional permissions than what is needed. To reduce attack vectors in function chains, they recommend implementing the least privilege principle, designating roles according to functions, and decreasing wildcard permissions (Marin et al., 2022a)

Function chaining has been discussed in more security feedback as a potential method of privilege escalation between services. It suggests splitting functions while making sure each gets the permissions that are really essential (Barringhaus, 2025; Joel, 2023). In serverless

security, proper sandboxing and secret management are essential. There are several significant risks associated with employing shared roles or without encryption environment variables.

2.9.4 Multi-Tenancy and Isolation

Multi-tenancy is an element of serverless platforms by default. Shared infrastructure can handle functions from several tenants (or users), and virtual isolation is the responsibility of the cloud provider. This raises the possibility of side-channel assaults, resource congestion, and loud neighbour effects.

Different isolation techniques are used by serverless computing platforms like AWS Lambda, Google Cloud Functions, and Azure Functions (e.g., microVMs like Firecracker, lightweight container sandboxes, or gVisor). None among these various approaches are capable of completely eliminating leakage risks or performance interference when dealing with high loads. Lightweight isolation methods (such microVMs and containers) may still have weak boundaries, which leaves them open to side-channel assaults and resource interference in multi-tenant environments, according to Li, Leng, and Chen (2021).

Similar situations have also shown performance isolation problems. The "noisy neighbour" phenomenon is confirmed by experiments conducted in containerised settings, which show that a heavy workload in one container can impair the responsiveness of co-located containers (Zhao et al., 2017).

In order to prevent breaches of data while offering access clarity, multi-tenant serverless applications—where a single React frontend serves many tenant environments—need to maintain proper isolation. Tenant-specific Firestore collections or structures, like the following, are frequently employed:

- Each document in this collection of top-tier **schools** (or **organisations**) represents a tenant and comprises subcollections specific to that tenant, such as **users**, **courses**, **resources**, etc. Tenant data is segregated into their own storage slice using this configuration. Firestore rules scoped under each tenant's document are utilised afterwards to enforce security (Lüdemann, 2020).

Particularly in multitenant React + serverless CMS systems, this per-tenant data partitioning makes logging and debugging more obvious by reducing the blast radius of permission issues and defining access paths.

A variety of new runtime micro-isolation techniques, especially WebAssembly-based runtimes, gVisor secure containers, Firecracker microVMs, and V8 isolates (such as those used by Cloudflare Workers), are being explored as possible lighter and more safe alternatives for conventional containers. For example, the VEE 2020 research "Blending Containers and Virtual Machines" demonstrates that while both Firecracker and gVisor run more kernel code than native Linux processes, the two offer greater isolation than native containers (Anjali et al., 2020).

According to a recent serverless edge computing evaluation, Cloudflare Workers' application of V8 isolates delivers a significantly reduced startup latency and much faster cold starts than microVM-based methods like Firecracker, albeit at an expense of runtime flexibility and isolation strength (Siidorow, 2024). According to studies, unikernels (such as Nanos and OSv) have quicker concurrent cold start performance than microVMs and containers, starting 100 instances in just under a second, while microVMs and containers took a few seconds to boot (Moebius et al., 2024).

2.9.5 Compliance, Governance, and Regulatory Constraints

Cloud-native applications are growing more concerned about data compliance, particularly those employed in regulated sectors like banking, healthcare, or education. Regional standards for data residency, logging, encryption, and access auditability should be met by serverless platforms.

The stateless, ephemeral invocation structure of serverless systems makes regulatory compliance—whether controlled by GDPR, HIPAA, or SOC 2—extremely challenging since it makes impossible to provide reliable information tracing, retention control, and secure elimination. Current cloud concepts lack accurate, end-to-end data flow tracing, which makes it problematic to guarantee regulatory compliance, as noted by Kunz et al. (2020). Additionally, Shastri, Wasserman, and Chidambaram (2019) emphasise that contemporary designs frequently violate GDPR anti-patterns such poor traceability and inadequate data lifecycle management. In reality, developers working on React-based frontends have to be extra careful not to unintentionally log personally identifiable information (PII) because log

statements can save emails along with other personally identifying information unless they are anonymised or overlooked, which makes compliance more difficult. Strict log retention, data categorisation, pseudonymization, and accurate observability pipelines are all essential to bring serverless systems into compliance with statutory requirements.

Complete activity tracking becomes possible by integrating several of AWS's compliance-enabling services, such as CloudTrail, X-Ray, and Config, into Lambda functions. In the same way, Firebase provides regional hosting, access logs, and data export restrictions to adhere to data protection regulations.

For data controllers attempting to adhere to regulatory requirements, serverless platforms typically do not include native, end-to-end data lineage tracking, which raises challenges. It becomes problematic to deliver useful answers to "who accessed what data, when, and how" when function executions are short and insufficiently documented. According to architectural literature, serverless systems need specially designed governance and auditing tools because relying only on cloud defaults is insufficient to meet requirements for compliance (Crudu, 2024).

Some practitioners recommend treating principles of governance like code, versioning them in combination with serverless function deployments, and enforcing them at runtime as a way to close these gaps. This method aids in guaranteeing that access controls, retention guidelines, and data flow rules remain the same throughout development cycles. The proactive implementation of governance rules into CI/CD pipelines improves auditability and minimises risk for controllers in circumstances such as regulated systems (e.g., under GDPR or HIPAA).

2.10 Developer Experience and Productivity in Full-Stack Serverless Applications

The developer experience (DX) which serverless architecture offers is just as crucial to its success in production environments as its scalability and effectiveness. The simplicity with which frontend developers can develop, test, and maintain backend logic, APIs, and integrations without extensive infrastructure knowledge is critical in terms of their productivity, particularly when working with frameworks like React. With a concentration on React-based full-stack applications, this section analyses the ways in which serverless development influences workflow design, learning curves, tooling ecosystems, and debugging approaches.

2.10.1 Learning Curve and Entry Barriers for Serverless Development

Developer-friendly serverless architecture is frequently advertised, particularly for teams that focus frontend development. However, new paradigms are introduced when old client-heavy or monolithic apps are swapped with function-based backend models.

At start, serverless platforms come with a relatively short learning curve, enabling developers to quickly set up essential functionalities. It is important to comprehend platform-specific concerns like cold starts, IAM role configuration, event-driven behaviour, and environment scoping, nevertheless, causes the curve to rapidly steepen as they progress.

This trajectory has been backed by multiple surveys and reports:

- According to the CNCF, a significant serverless pain factor identified by 28% of developers includes cold start latency.
- Though serverless abstracts away some infrastructure, it takes significant cloud-native expertise to effectively handle permissions, deployment intricacies, and asynchronous processing logic, according to commentary on developer cognitive responsibilities.
- The difficulty for developers going beyond simple scenarios is further increased by ecosystem-wide challenges, such as the dearth of reliable local emulation tools and the complexity of debugging distributed pipelines. (Cesar, 2021; Chaudhary, 2020; Nguyen, 2024; Siidorow, 2024)

Inconsistencies in mental models produce complications when integrating serverless backends, especially for React developers. Whereas serverless necessitates externalising business logic into individual backend functions, React supports component-driven, client-side logic. This differentiation often results in context shift between codebases and fragmented workflows.

To reinforce the separation of responsibilities and accelerate co-development of frontend and backend logic, multiple thought leaders recommend matching the modular structure of React components with comparable serverless functions—a framework we could possibly refer to as "Function Component Alignment." This approach promotes a clear mapping between UI elements and their backend actions, which might improve maintainability and developer

comprehension even though it hasn't been completely studied yet.(Hefnawy, 2016; Paghar, 2024; Shah, 2024)

2.10.2 Tooling Ecosystem and DX Optimisation

Developer experience is significantly shaped by tools. Setup barrier is considerably reduced by platforms like Vercel, Netlify, and AWS Amplify, thanks to "zero-config" onboarding and easy Git integration.

Due in large part to Netlify's integrated CI/CD, streamlined function configuration, and one-click deployments, practitioners say that developer experiences with Netlify—when delivering React apps alongside serverless functions—are considerably easier than manual AWS CLI-based deployments. Vercel views backend logic as a first-class component of the frontend codebase with its integration of Next.js, including support for SSG, SSR, ISR, and edge functions. This enables greater cohesiveness and more efficient full-stack workflows (Drasner, 2021; Savants, 2025, p. 5; Vercel, 2025a; Voss, 2021).

Platforms like Vercel make it easier to deploy serverless and React apps, but they frequently hide essential details about configuration with their abstraction. For instance, developers can't configure memory or CPU restrictions in code using Vercel functions; instead, these are controlled automatically through the dashboard's broad tiers (e.g., 2 GB/1 vCPU by default, upgradeable only at the Pro/Enterprise level) (Vercel, 2025b, 2025c). Such restrictions, which show up as more cold starts, throttling, or failures during load tests, typically are only discovered by developers after deployment, undermining confidence in automated setups.

Teams generally employ techniques that provide explicit settings to fix this lack of visibility:

- **Serverless Framework**: Using serverless.yml, it allows multi-environment deployments, centralised function definitions, and explicit resource settings.
- **Firebase Emulator Suite**: Decrease deployment uncertainty by supporting local authentication and Firestore testing.
- Declarative infrastructure modelling with complete control over runtime, IAM, and resource settings are provided by AWS SAM/CDK.
- **Live reloaders**, such as Nodemon and Vite, enable seamless front-end-backend development cycles and instant feedback.

According to study, Bilal et al. (2021) contend that serverless platforms' lack of fine-grained resource establishing impedes cost effectiveness and performance optimisation, suggesting systems that separate CPU and memory allocation would benefit developers.

2.10.3 Debugging Workflows and Troubleshooting Challenges

Comparing serverless environments to monolithic or containerised systems, debugging is notably harder. Because functions are stateless and work in transient settings, they can not be debugged using conventional techniques like step-by-step breakpoints and persistent logs.

Practitioners claim that in serverless systems, where execution processes can be hidden by scattered invocation patterns and transitory function lifecycles, observability—rather than traditional debugging—is crucial. For fault diagnosis, structured logging, event replay, and distributed tracing (using programs like OpenTelemetry or AWS X-Ray) are essential. While Fred & Olasehinde (2019) examine the more general challenges of achieving observability through logs and tracing, Borges et al. (2021) show how distributed tracing enhances fault localisation in serverless compositions. Additionally, stateless executions and fragmented observability make failure detection hard unless powerful tracing pipelines are in place, as demonstrated by visual research on anomaly detection.

Correlation IDs and trace headers play an important role in React/serverless apps since a single user interface event can set off several asynchronous backend operations. Some teams employ trace propagation models, in which unique trace IDs are injected at the frontend and transmitted along the serverless chain, to increase visibility. This is furthered by commercial tools like Thundra, Dashbird, and Lumigo, which provide real-time error tracking, visual execution maps, and latency analysis, supporting developers in lowering down on debugging time and boosting confidence in serverless deployments (Borges et al., 2021; Datadog, 2025; Plößer, 2021).

2.10.4 Developer Productivity: Metrics and Empirical Findings

Modular codebases, fast deployment, and lower infrastructure overhead are frequently connected to productivity in serverless development. Such advantages, however, are not equally distributed all through team configurations and experience levels.

Comparing serverless services to container-based methods like Kubernetes or EC2, it turns out that the former method reduced infrastructure provisioning and deployment overheads by

20–30% (Eivy and Weinman, 2017; Roberts, 2018). Developers cited up to a 15% increase in debugging time during cross-team integrations, suggesting that these perks came at the expense of more complex debugging (Taibi et al., 2021). Colocating backend APIs within the same project, such as through Next.js API routes or colocated Lambda folders, maximises productivity for frontend-heavy teams working with React. This is due to how it minimises context-switching and allows for the shared use of TypeScript types, utilities, and test suites (Netlify, 2025b; Vercel, 2025a).

By offering ready-to-use structures for authentication, hosting, and CI/CD, template-based scaffolding tools—like the AWS Amplify CLI, Firebase CLI, or Create React App combined with Netlify CLI—speed up the development of MVPs. These tools assist teams create a uniform project architecture and increase early-stage productivity. Although scaffolding frameworks remove repetitive setup procedures, developers may be able to boost programming efficiency by up to 25 percent, according to studies (Mărcuță, 2024). However, as initial development gets going, problems like incorrect IAM roles, environment drift, and irregular versioning of API contracts, in addition to poor backend documentation, frequently cause productivity to decline.

One specific danger is API contract misalignment: even in systems with robust type and anticipated discipline in the process, 27% of integration issues were linked to mismatched API definitions in a case study of an enterprise team making use of contract-driven development (Specmatic, 2023). By boosting consistency, documentation, and auto-generated client–server code alignment, technologies like Swagger/OpenAPI, Postman collections, GraphQL schemas, or tRPC-style type-safe API techniques can help mitigate these problems.

2.10.5 Collaboration and Role Separation

Collaboration between frontend and backend engineers is crucial in full-stack teams. Clearer division of roles is made attainable by serverless architecture; backend developers create serverless APIs and oversee permissions, while frontend teams concentrate on React, UI logic, and token management.

Full-stack roles, wherein developers handle both the user interface and backend APIs, are common among startups and small teams. This architecture is supported by serverless computing, which lowers infrastructure overhead while making backend maintenance easier for teams that prioritises the frontend. 63 percent of the 100 practitioners surveyed said that

they were full-stack engineers, referring to serverless as a factor in the convergence of roles (Taibi et al., 2021). Centralising function monitoring, cloud console access, or deployment benefits, however, causes problems because it could restrict developer autonomy and impede iteration. Internal developer portals for centralised logs, metrics, and API schemas; role-based IAM boundaries for secure access; and GitOps-based workflows for shared visibility are numerous instances of recommended practices. Platforms which offer visual interfaces for controlling both frontend and backend resources, like AWS Amplify and Firebase Console, significantly reduce this friction (Mărcuță, 2024).

2.11 Synthesis, Research Gaps, and Future Directions

2.11.1 Synthesis of Key Themes

As the literature analysis above shows, serverless architecture has developed into a well-established yet quickly evolving paradigm with major implications for full-stack React apps. Performance, scalability, developer experience, and cost-effectiveness all show up as important topics.

First, by abstracting infrastructure management and dynamically allocating resources, serverless computing offers significant scaling benefit. This elasticity is especially useful for React-based applications because tasks like content updates and real-time collaboration frequently shift in response to actions taken by users. Automatic scaling lowers operational complexity, according to studies, making serverless appealing to both small teams as well as big corporations (Baldini et al., 2017; Jonas et al., 2019a).

Second, An appealing value proposition is introduced by the serverless cost model, which is based on execution time and request volume rather than static resource allocation. Serverless platforms frequently outperform traditional virtual machines (VMs) or container clusters when it comes to of efficiency and price for spiky and unpredictable workloads, according to comparative studies of AWS Lambda, Azure Functions, and Google Cloud Functions (Lloyd et al., 2018; Shahrad et al., 2020). This benefit, however, wanes for constant high-throughput applications, hence workload profiling is essential prior to using serverless at scale.

Third, serverless solutions include benefits and drawbacks in terms of developer experience (DX). React developers are able to develop end-to-end systems with less backend knowledge thanks to platforms like Firebase and Vercel, which offer close connectivity with frontend frameworks. However, actual data reveals that while working with serverless applications,

developers often have difficulties with debugging, inconsistent local-to-cloud settings, and a shortage of useful observability tools (Wen et al., 2022). These results are additionally backed by a deeper systematic analysis, pointing out that deployment complexity, cold-start delay, and inadequate monitoring continue to be major challenges, especially when developing intricate distributed systems (Eismann et al., 2022; Wen et al., 2022).

Lastly, even though React offers a strong client-side ecosystem, there are some architectural trade-offs when pairing it with serverless backends. Clearer separation of duties can be made achievable by aligning React components with the corresponding serverless methods, that could improve modularity and maintainability (Taibi et al., 2021)However, studies also shows that this method could result to further fragmentation, context switching, and hidden technical debt, especially among full-stack teams that oversee several separate departments (Eismann et al., 2022; Jonas et al., 2019a).

2.11.2 Identified Research Gaps

2.11.2.1 Empirical Studies on Full-Stack Use Cases

Although the majority of current research focus on either frontend frameworks like React (Baldini et al., 2017) or serverless backends (Jonas et al., 2019a; Lloyd et al., 2018), few studies look at both as a single, integrated full-stack approach. As a result, there is little empirical data comparing the productivity, maintainability, and cost-effectiveness for end-to-end developers when using React alongside with serverless functions to other stacks like MERN or JAMstack that have no serverless functions.

2.11.2.2 Standardised Benchmarks for Serverless Performance

There is no established standard for assessing performance in diverse workloads, despite being the case that there are a number of benchmarking tools for serverless platforms (Eismann et al., 2020; McGrath and Brenner, 2017). Comparability is hampered by the large variation in cold start delay, concurrency stress test, and cost model metrics between investigations. Particularly there isn't just one benchmark that targets full-stack applications that integrate React.

2.11.2.3 Long-Term Maintainability and Technical Debt

Long-term technical debt is little acknowledged, despite serverless being frequently pitched as cost-effective. The complexities of handling distributed logic doubles with the number of

functions. Though they frequently fall short of offering quantitative proof across big projects, empirical studies draw attention to these issues, such as increased cognitive burden, configuration sprawl, and debugging difficulty (Eismann et al., 2022; Jonas et al., 2019a; Lloyd et al., 2018) To evaluate maintainability and the cumulative effect of function proliferation, additional long-term research is required.

2.11.2.4 Security and Privacy in Frontend-Heavy Architectures

Cloud infrastructure misconfigurations, including too permissive IAM roles or incorrectly configured storage buckets, frequently become the subject of security investigations (Wen et al., 2025, 2022)Few studies, however, look into vulnerabilities in React + serverless systems that are specifically caused by frontend-driven logic. There is a need for more study on full-stack serverless security since problems like mismatched access policies, insecure token management, and API exposure are still not adequately addressed in the literature.

2.11.2.5 Developer Experience Metrics

Instead of using systematic quantitative indicators, the majority of DX research depends on anecdotal evidence or small-scale surveys. For example, a generic methodology to evaluate developer experience is proposed by Greiler, Storey, and Noda (2023), however serverless ecosystems weren't included in its application. Similarly, developer onboarding, debugging efficiency, and productivity are not sufficiently examined in empirical research by Baldini et al. (2017), who mainly focus on performance and cost trade-offs. Standardised DX benchmarks would allow for helpful comparisons between platforms like Firebase, Netlify, and AWS Amplify.

2.11.3 Implications for Practice

The synthesis also yields several actionable implications for practitioners:

- Workload Profiling First: Before switching to serverless, organisations must decide if workloads are constant or irregular because workload characteristics have a significant impact on cost-effectiveness (Castro et al., 2019; Villamizar et al., 2017).
- Observability-First Design: To address the unresolved monitoring and observability issues identified by Baldini et al. (2017), distributed tracing and structured logging ought to be incorporated from the beginning to lessen debugging difficulties.

- Integrated Toolchains: In line with Greiler, Storey, and Noda's (2023) focus on enhancing the developer experience, co-locating frontend and backend logic within a single repository, facilitated by frameworks like Next.js or Amplify, can lower friction and increase productivity.

These implications emphasise that in order to get the full benefits of serverless adoption, organisational and workflow strategies must be implemented in line with technical considerations.

Chapter 3: Methodology

3.1 Research Design

A typical full-stack application was developed utilising serverless architecture and then evaluated under carefully monitored experimental conditions as a part of the current study's prototype-based case study design. This design is in line with the design science paradigm frequently utilised in computing research since it permits the creation of an artefact alongside its empirical evaluation (Hevner and Chatterjee, 2010).

Since automated benchmarking tools have been utilised to thoroughly measure latency, throughput, request success rates, and scalability, the methodology is essentially quantitative. Given the necessity to comprehend trade-offs between cost, latency, and scalability, performance-focused assessments of serverless systems have proliferated in the scholarly literature (Eismann et al., 2020; Jonas et al., 2019a; Shahrad et al., 2020). Simultaneously, the study integrates qualitative insights about developer experience (DX), specifically concerning workflows associated with deployment, monitoring, and debugging. These observations provide contextual information on how serverless platforms affect software development techniques, but they are not meant to be a comprehensive qualitative research.

Therefore, the best way to express this approach is as an empirical case study that is driven by quantitative analysis and includes some qualitative interpretation. While accepting that developer experience is an essential element of assessing serverless systems, this strategy avoids overstatement of being a mixed-methods architecture.

3.2 Prototype Development

The study's artefact was a To-do application that was built with AWS Lambda for serverless backend logic, DynamoDB for persistence, AWS API Gateway for request routing, and React for the frontend.

The to-do application selection was strategic. The program captures the common CRUD (Create, Read, Update, Delete) activities that predominate in enterprise workloads, despite its simplicity. These workloads serve as an excellent starting point for testing serverless architectures since they are latency-sensitive and need recurrent requests to backend services (Lloyd et al., 2018). In addition, this solution avoided the hassle of creating a completely custom system while enabling performance evaluation under user-like interaction scenarios.

Leveraging the on-demand provisioning and auto-scaling features inherent to serverless platforms, the prototype was set up in the AWS cloud. Infrastructure as Code (IaC) templates were utilised to automate deployment, ensuring consistency and reproducibility throughout test runs.

3.3 Evaluation Metrics

To evaluate how suitable is serverless architecture for scalable and cost-effective full-stack applications, the below mentioned metrics are being defined:

1) Latency: It is defined as the total time takes to process the request and return the response to a client. This calculation comprises the end-to-end journey through AWS API Gateway, Lambda functions, and DynamoDB operations. Latency is recorded in terms of :
    i) Minimum, Maximum, and Average response time.
    ii) Percentile value at p50 (median),p95, and p99, which are commonly used indicators for user experience and check latency behaviour (Shahrad et al., 2020).
2) Scalability: It is defined as the system’s ability to maintain the stable latency and the amount of data it can process in a specific time period under increased request loads. Scalability is assessed through:
    i) Request rates (10 req/s up to 50 req/s).
    ii) Number of concurrent virtual users (vusers).
    iii) Throughput (requests that has been successfully processed per second).
    iv) Ratio of successful to failed requests.

3) Cost Efficiency: The cost was estimated rather than measured directly because serverless systems are compensated based on the number of requests and execution time. Expected costs under various workload scenarios were determined using DynamoDB's pricing model (per read/write operation) and AWS Lambda's cost model (per 1 million requests and every GB-second of computation time). This makes it possible to weigh the expenditures against the advantages of scalability (Eivy and Weinman, 2020).

This multifaceted assessment approach made sure that the methodology addressed concerns with developer usability and cost besides technical performance.

3.4 Data Collection Methods

3.4.1 Latency Measurement

Artillery, an open-source load testing toolset, was used to measure latency. The minimum, maximum, mean, median, and high-percentile (p95 and p99) response times have been collected for each test. The technique adheres to accepted procedures for assessing serverless systems, where tail latency is particularly important for applications which interact with users (Baldini et al., 2017).

The two phases conducted were:

1) Warm-up phase(60 seconds): Here the traffic is being gradually increased up to 10 requests per second. This made sure the AWS Lambda functions were “warmed up” and the effect of cold starts dominating the data is reduced.
2) Sustained load phase (120 seconds): Constant traffic up to 50 requests per seconds, through which observation of steady-state latency and throughput is enabled.

The study used Dotcom-Monitor as an additional assessment tool in addition to Artillery, which was used to test server-side latency and scalability and simulate regulated traffic. Synthetic end-to-end monitoring from distributed places was made possible by Dotcom-Monitor, giving insight into the actual user experience. Typical user journeys, including logging onto the program, making to-do lists, editing, and deleting, were replicated by test scripts.

This combination made sure that performance was assessed from two angles: client-perceived responsiveness across networks and regions (as recorded by Dotcom-Monitor) and system scalability and throughput under stress (as recorded by Artillery). The methodology

provides a broader assessment of latency, reliability, and user experience in the serverless architecture by combining controlled and external vantage point testing.

3.4.2 Scalability Testing

In order to imitate load escalation, artillery was also used, with increasing virtual users and request rates. Throughput, successful vs unsuccessful queries, and system stability under increasing load were amongst the metrics. These findings made it possible to investigate AWS Lambda's horizontal scaling behaviour, which is a characteristic that distinguishes serverless systems.

3.4.3 Cost Estimation

In an attempt to replicate load escalation, artillery was also employed, with rising virtual user counts and request rates. Throughput, successful vs unsuccessful queries, and system stability under increased load were among the metrics. These findings made it possible to examine AWS Lambda's horizontal scaling behaviour, which is a feature that distinguishes serverless systems.

3.5 Validity and Reliability

The following steps were carried out to increase the results' robustness:

- Internal Validity: Three rounds of each test scenario were carried out and the averages are given. Outliers (resulting from fleeting network oscillations or irrelevant background activity) were examined and removed if they were considered non-representative.
- External Validity: The results of this study are applicable to a broader class of CRUD-heavy applications, which represent a significant amount of enterprise workloads, despite being developed on a to-do app. This is consistent with the benchmarking methods used in earlier serverless studies (Eismann et al., 2020; Jonas et al., 2019a).
- Reliability: Reproducibility was guaranteed by managing testing scripts and configurations under version control. IaC templates were used to automate deployment, reducing the chance of configuration drift between test runs.

3.6 Ethical Considerations

Since there were no human subjects in this investigation, there were little ethical issues. The primary ethical consideration was how to use cloud resources responsibly. The study decreased the environmental impact of excessive energy usage and prevented needless financial expenses by limiting the length of tests and calculating costs using modelling.

Chapter 4: System Design and Implementation

4.1 Architecture Overview

The To-do application follows a serverless architecture which is hosted on AWS platform. It uses AWS components which are already described in methodology. The basic structure behind the application is comprises of react frontend stored in S3 bucket, interaction with backend functionality via API gateway, backend logic to handle all operations by AWS Lambda function which performs CRUD operations on Dynamo DB tables.

Figure 4 System architecture Figure 5 AWS Architecture Overview gives high level system architecture overview:

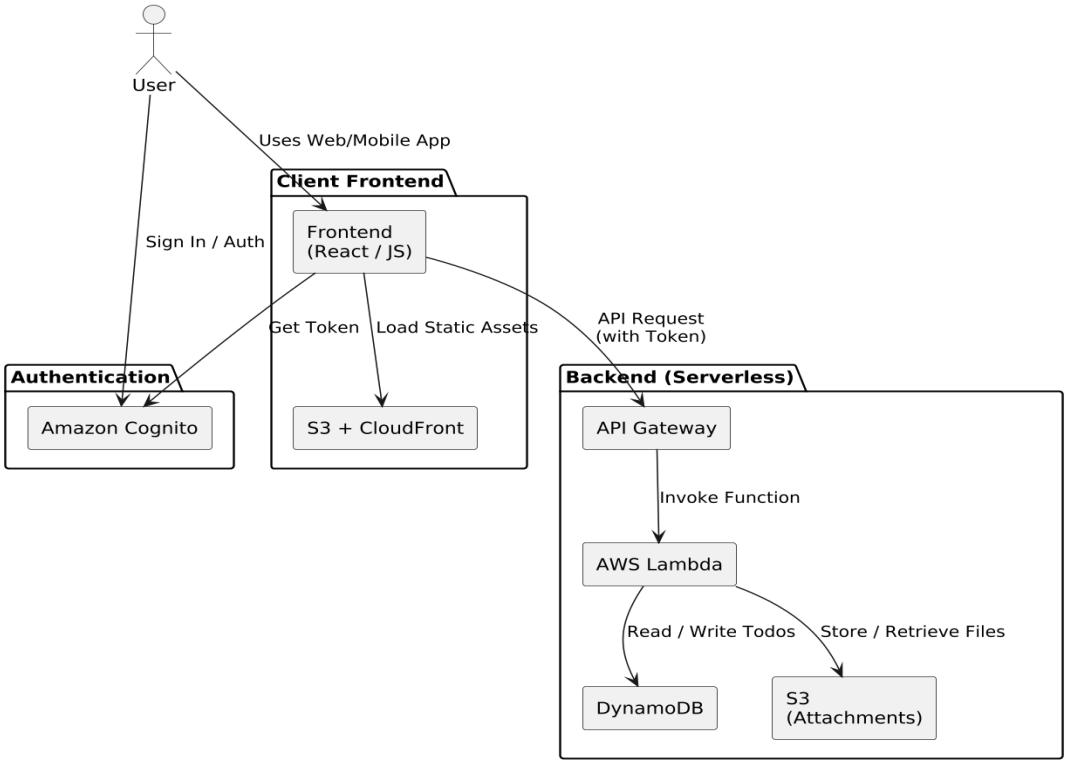


Figure 4 System architecture

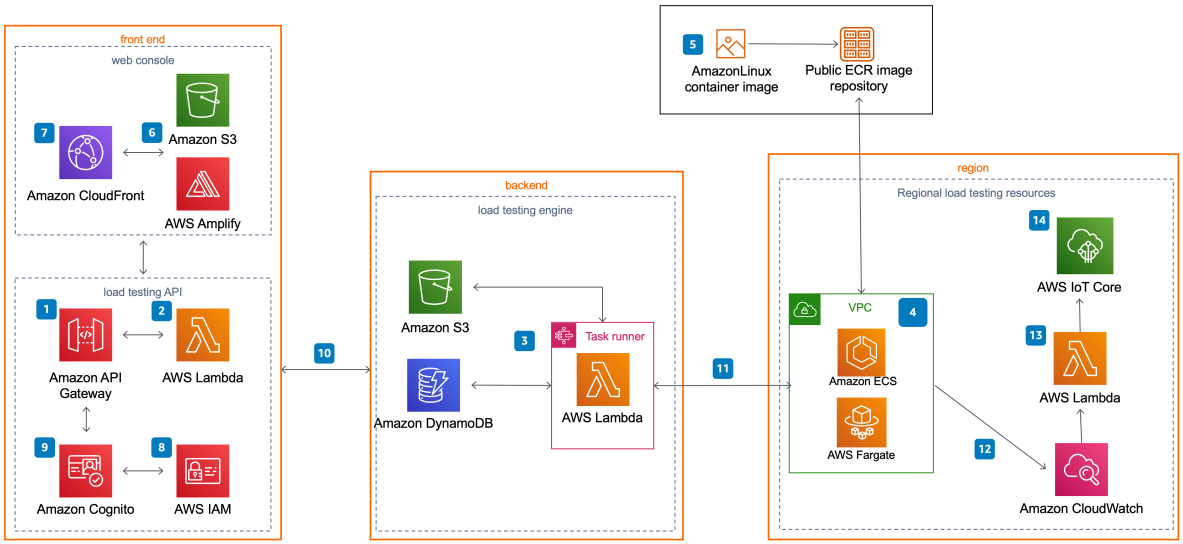


Figure 5 AWS Architecture Overview

4.2 Frontend – React

The user interface consists of a React-based single-page application that enables users to create new tasks, toggle their completion status, and remove them entirely. The interface responds dynamically to state changes managed by React, maintaining synchronization with the server through REST API requests to AWS API Gateway.

Below is screen-shot of application interface which displays marked tasks. System UI uses double click to edit the task and search bar to add as well as search existing task. The cross mark present in front of every task can be used to delete the task.

Code of front-end react is added in appendices under appendix A app.jsx. Basic code routing depending upon API call is also added in same file. Figure 6 Application Interface shows interface of application.

Figure 6 Application Interface

4.3 Backend Serverless functions

AWS cloud provide serverless options like Lambda, Dynamo DB. AWS Lambda functions has been used in two application. One for TO-DO application and another for load testing which includes all the metrices for testing as shown in Figure 7 AWS Lambda applications

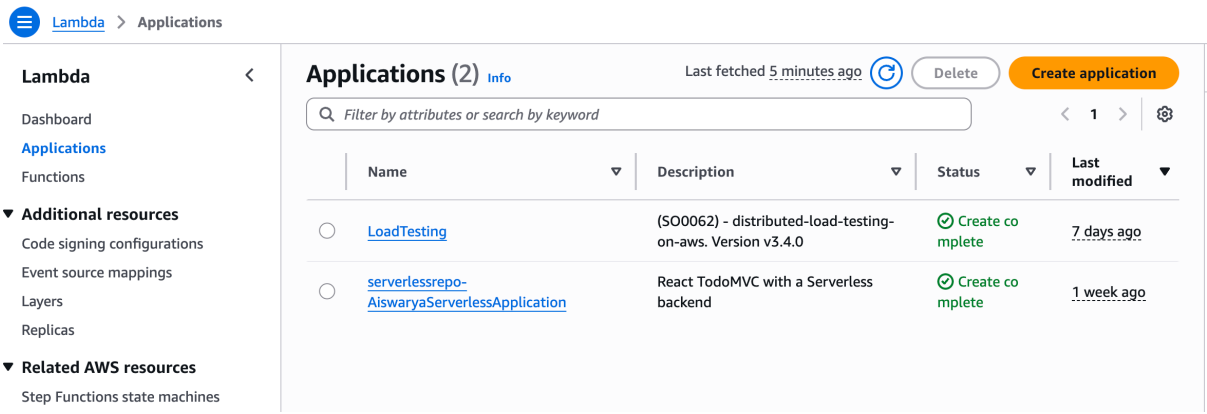


Figure 7 AWS Lambda applications

AWS Lambda functions has been used as backend brain of serverless app, here it is responsible for processing and interacting with other services. Ex handling CRUD operations based on specific triggers and API endpoints as shown in Figure 9 Lambda API Gateway and Figure 8 API call codesnippet from utils.js file shows the API call handling once triggered by lambda function.

```
// Post to the API to store todos
store: function (data) {
    fetch(BASE_PATH + 'api/todos', { method: 'POST',
        body: JSON.stringify(data)
    }).then(function(response) {
        return response.json();
    }).then(function(json) {
        console.log('saved');
    }).catch(function(err) {
        console.error(err);
    });
},

// Fetch todos from the API
load: function(callback) {
    fetch(BASE_PATH + 'api/todos').then(function(response) {
        return response.json();
    }).then(function(json) {
        return callback(null, json);
    }).catch(function(err) {
        return callback(err);
    });
},
```

Figure 8 API call code

AWS Lambda is automatically scaling based on incoming requests which is discussed in results and calculations. Ideal state cost nothing unlike traditional applications ex. Figure 10 AWS Lambda Functions.

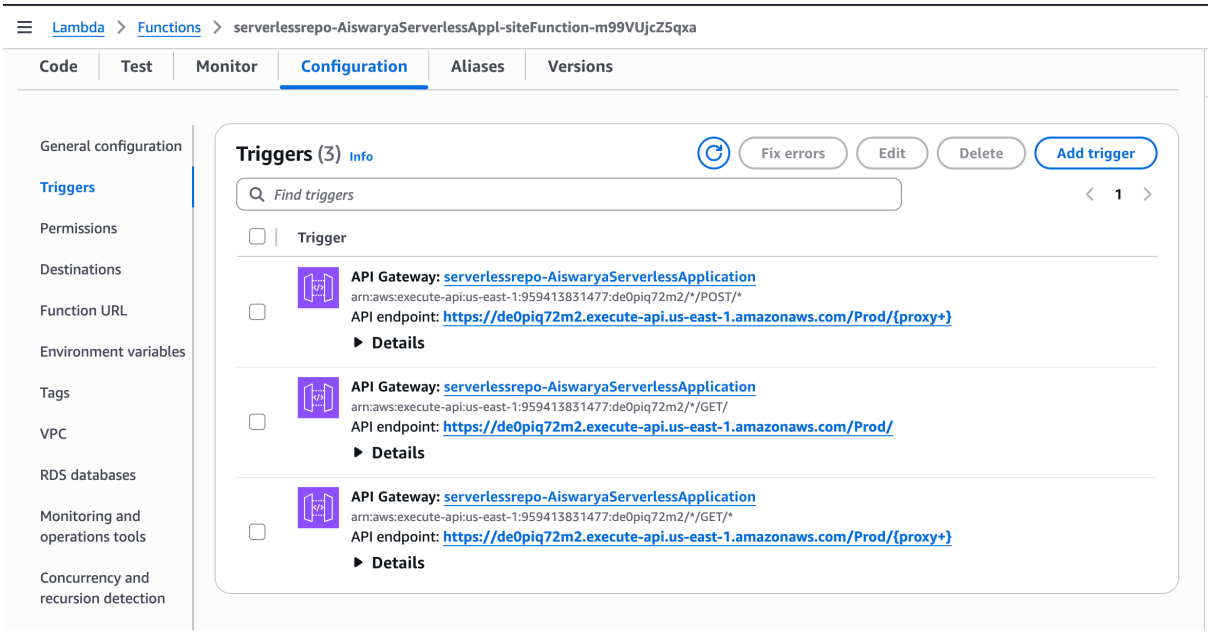


Figure 9 Lambda API Gateway

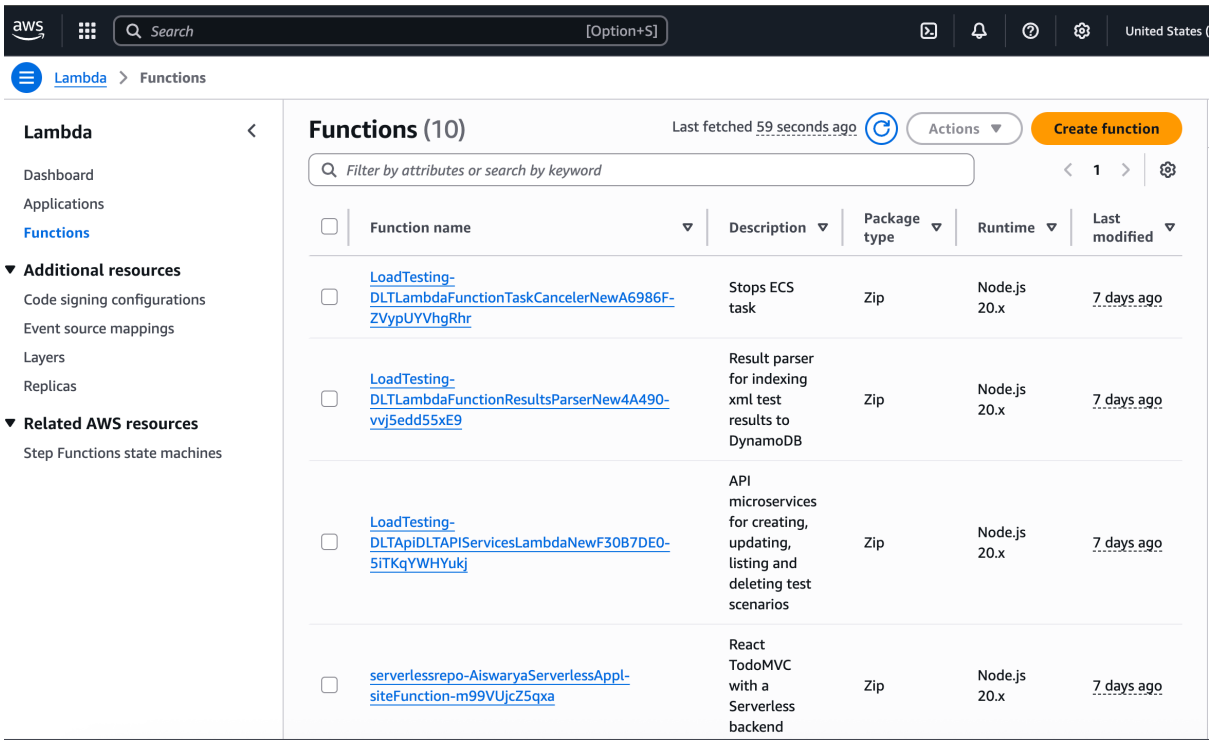


Figure 10 AWS Lambda Functions

4.4 Deployment Strategy

To deploy the serverless To-Do application, **AWS Serverless Application Model (AWS SAM)** is used, which simplifies the definition and deployment of serverless resources. SAM allows the entire infrastructure including API Gateway endpoints, Lambda functions, and DynamoDB tables to be described in a YAML template and deployed as a single unit.

```yaml
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31
Description: React TodoMVC with a Serverless backend
Globals:
  Api:
    BinaryMediaTypes:
      # The ~1 will be replaced with / when deployed
      - '*~1*'
Resources:
  siteFunction:
    Type: AWS::Serverless::Function
    Properties:
      Description: React TodoMVC with a Serverless backend
      Handler: bundle.handler
      Runtime: nodejs20.x
      CodeUri: todo.zip
      Policies:
        - DynamoDBCrudPolicy:
            TableName:
              Ref: todoTable
      Timeout: 10
      Events:
        root:
          Type: Api
          Properties:
            Path: /
            Method: get
        getProxy:
          Type: Api
          Properties:
            Path: '/{proxy+}'
            Method: get
        postProxy:
          Type: Api
          Properties:
            Path: '/{proxy+}'
            Method: post
      Environment:
        Variables:
          TABLE:
            Ref: todoTable
  todoTable:
    Type: AWS::Serverless::SimpleTable
```

4.5 Testing application configuration

4.5.1 Load testing with Artillery

To evaluate the performance and scalability of the serverless To-Do application, **Artillery** was integrated a modern, open-source load testing and functional testing toolkit specifically designed for testing backend APIs and serverless architectures. Artillery was employed to simulate HTTP traffic to the deployed AWS API Gateway endpoints connected to Lambda functions. This implementation allowed for early detection of potential bottlenecks, cold start

issues, and performance degradation under load before full-scale evaluations were conducted in Chapter 5.

Rationale for Artillery Selection Artillery was selected as the load testing framework due to its optimization for serverless and cloud-native environments. The tool offers a lightweight configuration approach through YAML or command-line interface and delivers comprehensive performance metrics including:

- Request throughput measurements (requests/second)
- Response time analytics (mean, median, p95, p99)
- Error rate tracking and failed request analysis
- Virtual user session duration monitoring

Workflow Integration Process Test configurations were developed using YAML format and executed through terminal commands via artillery run. The testing scripts incorporated multiple phases including warm-up and sustained load scenarios to replicate authentic usage patterns. Performance data was captured both in local logs and transmitted to Artillery's web-based dashboard to facilitate comprehensive result analysis.

```
config:
  target: 'https://de0piq72m2.execute-api.us-east-
1.amazonaws.com/Prod/#/'  # Replace with your API Gateway URL
  phases:
    - duration: 60  # Duration of the load test in seconds
      arrivalRate: 10  # Number of requests per second
      name: "Warm-up phase"
    - duration: 120  # Second phase, simulating sustained traffic
      arrivalRate: 50  # Requests per second (increase based on load)
      name: "Sustained traffic"
    - duration: 30  # Third phase, heavy load
      arrivalRate: 100  # High requests per second for peak load
      name: "Peak load"

scenarios:
  - flow:
      - get:
          url: "/POST"
```

Through the early integration of Artillery during the design and development phases, the application underwent testing under conditions that closely simulated real-world usage scenarios, thereby ensuring the serverless infrastructure's capability to manage concurrent requests without introducing latency or generating errors.

4.5.2 Load and Stress Testing with Dotcom-Monitor (LoadView)

Alongside Artillery, which was used to simulate controlled traffic and measure server-side latency and scalability, the study also employed Dotcom-Monitor as a complementary evaluation tool. Dotcom-Monitor enabled synthetic end-to-end monitoring from geographically distributed locations, providing visibility into the *real-world user experience*. Test scripts were configured to replicate typical user journeys, such as logging into the application via Amazon Cognito, fetching the to-do list through the API Gateway, creating tasks, and attaching files stored in S3.

Screen recording and multi-request submission were performed using the Dotcom Monitor LoadView application as shown in Figure 11

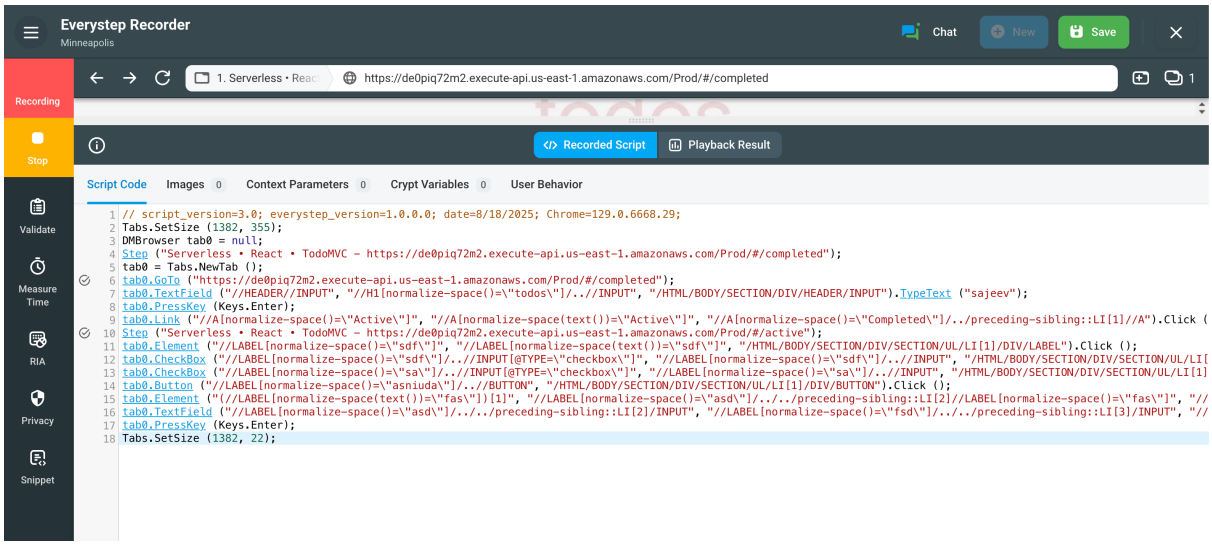


Figure 11 Dotcom-Monitor testing window

Chapter 5: Results

5.1 Overview of Experiments

This chapter discuss about the empirical results from two complementary tools:

1) Artillery: Using staged traffic profiles (low -> extreme), latency distributions, throughput, and error rates are measured with controlled backend load testing.
2) Dotcom-Monitor: Simulated end-to-end user experiences that have been carried out from an external location (Minnesota, USA) in order to replicate real-world, outside-in performance.

5.2 Backend Load Testing (Artillery)

5.2.1 Summary of Results

Stage	Requests/sec	Responses	Errors	Avg Latency (ms)	p95 Latency (ms)
Low Load	115	156	0	68.0	69
Medium Load	8,440	83,678	0	68.2	69
High Load	29,984	299,731	0	69.6	74
Extreme Load	54,754	545,921	0	96.7	156

Tabelle 1 Result Summary

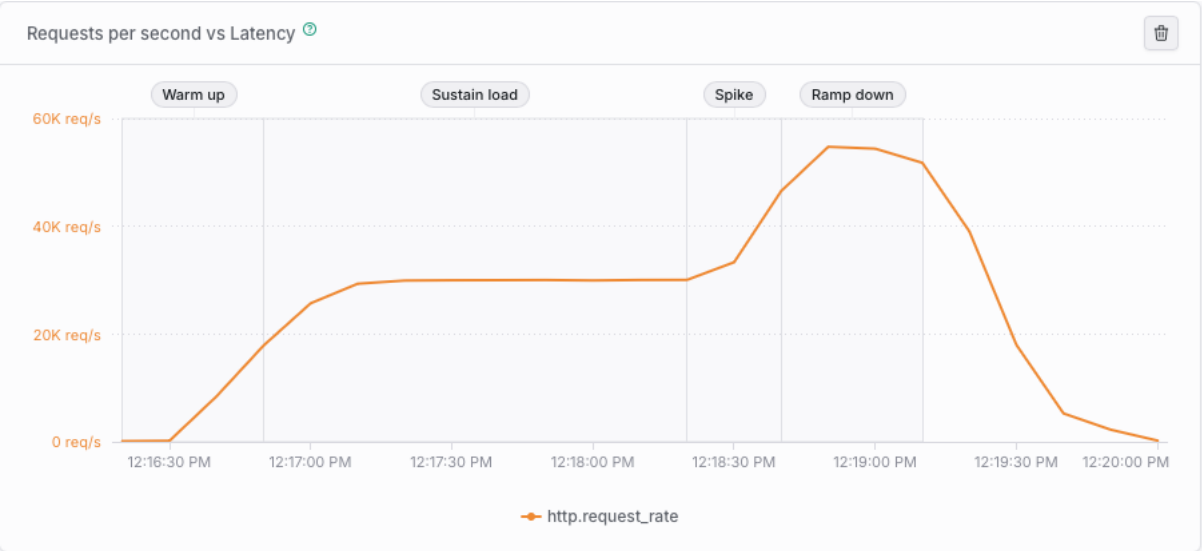


Figure 12 Artillery "Throughput vs Latency" chart

Throughput vs Latency Curve : scale that is almost linear until the High Load stage; with Extreme Load, there is a noticeable increase in mean and tail latency as shown in Figure 12.

Latency Distribution: Through High Load, p95/p99 remain low and tight; in Extreme Load, p95 expands to ~156 ms, with exceptions at ~470 ms as shown in Figure 13.

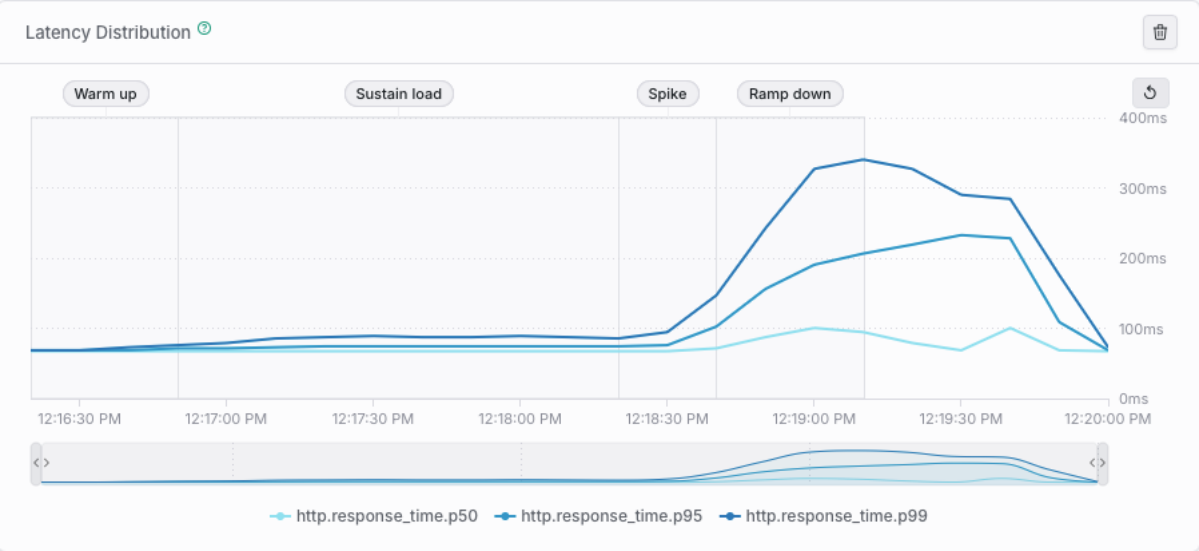


Figure 13 Artillery latency distribution (p50/p95/p99) chart

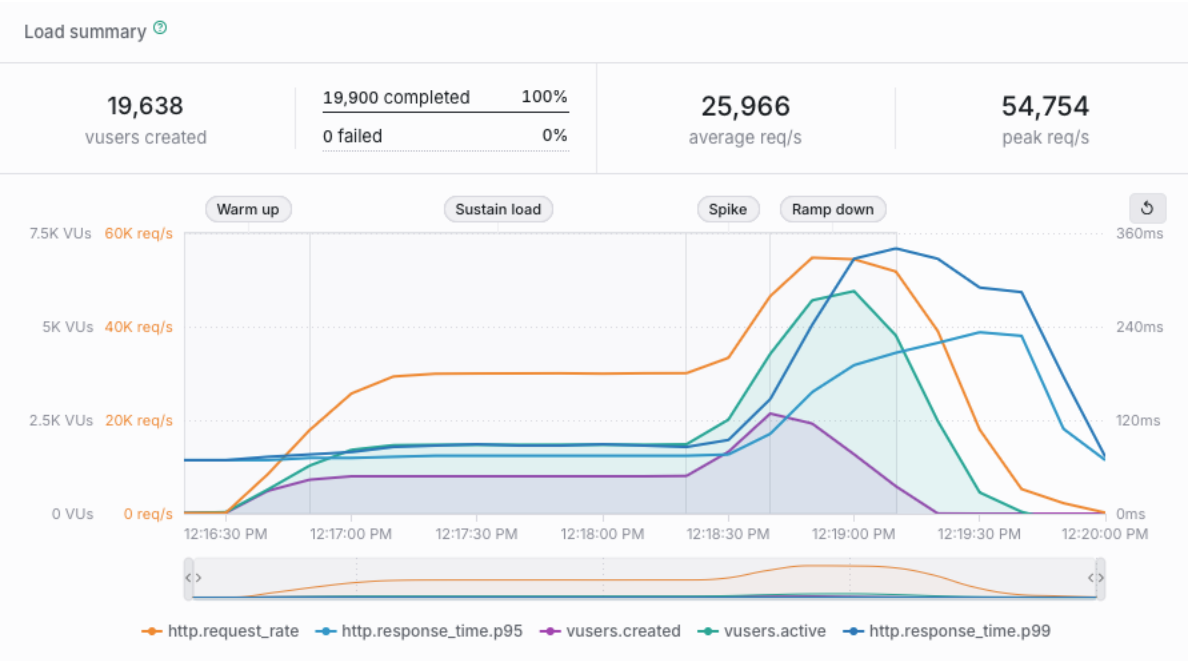


Figure 14 Artillary Load Summary Chart

Observations:

- Scalability: With only slight latency inflation (≈70 ms), the backend grew smoothly to approximately 30k requests per second as shown in Figure 14.
- Stress behaviour: For several CRUD workloads, overall mean latency increased to ~97 ms and p95 to ~156 ms at ~55k req/s.
- Reliability: Strong autoscaling and reliable error handling under load are indicated by 0 errors all over more than 1.0 million responses.

5.3 End-to-End Performance (Dotcom-Monitor)

5.3.1 Dotcom-Monitor Test (Moderate Load)

Metric	Result
Test Duration	2 min
Max Virtual Users	10
Total Sessions	21
Completed Sessions	21
Failed Sessions	0
Errors	0
Average Response Time	24.8-28.6 s
90% Response Time	27.4 s

Tabelle 2 Dotcom Monitor Result (Moderate Load)

Intrepretation: User-perceived delay is approximately 25–29 seconds, which is significantly greater than backend latencies; perfect reliability (100% completion). makes recommendations for possible first-time cold paths along with external path factors (network hops, TLS, SPA load time, Cognito redirection, and CDN edge behaviour) { ref Tabelle 2 }.

5.3.2 Dotcom-Monitor Test (Higher Load, Failures)

Metric	Result
Test Duration	3 min 36 s
Max Virtual Users	9
Total Sessions	12
Completed Sessions	4
Failed Sessions	8
Errors	4
Average Response Time	73.36 s
90% Response Time	74.10 s

Tabelle 3 Dotcom-Monitor Test Results (High Load)

Interpretation: 66% of sessions fail, and the average latency is extremely high (>70 s). Possible reasons include throttling at a particular tier, repetitious cold pathways accompanied by login redirects, or synthetic client saturation {ref Tabelle 3}

5.3.3 Dotcom-Monitor Stress Report

Metric	**Result**
Test Duration	2 min
Start -> End	00:14:49 → 00:17:25
Max Virtual Users	10
Total Sessions	21
Success Rate	100%
Average Response Time	26.34 s
90% Response Time	24.82 s
Min / Max Response Time	27.41 s / 28.60 s
Load Injector	US (Minnesota)
Injector CPU	Up to > 80%

Tabelle 4 Dotcom-Monitor Stress Result

Interpretation: Despite 100% session success, end-to-end latency is consistently around 26 s. Client-side overhead is hinted at by injector CPU >80%. Perceived time-to-interaction tends to be dominated by login + SPA boot + API round-trips + CDN fetches, as seen by the consistent ~25–29 s range across two successful tests {ref Tabelle 4 }.

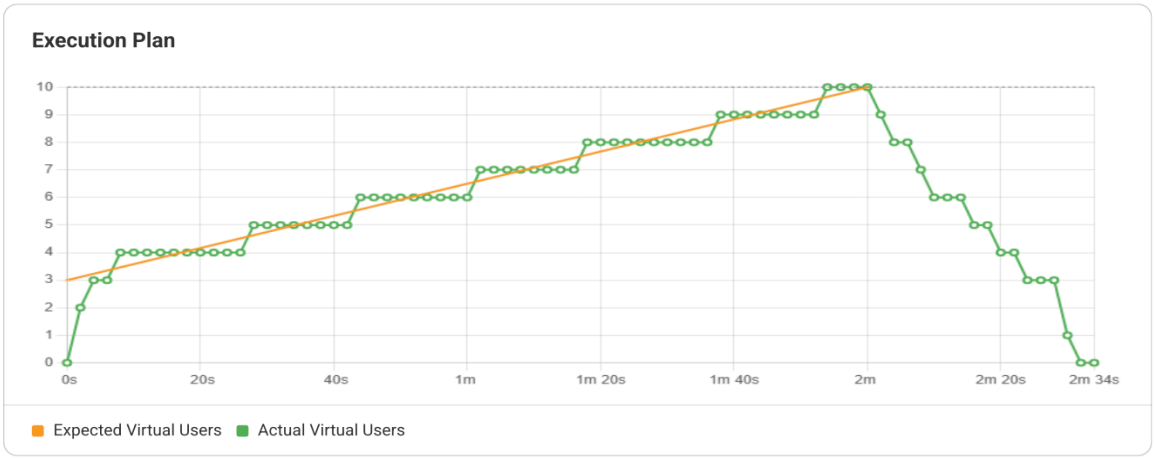


Figure 15 Dotcom-Monitor execution plan from the Stress Report

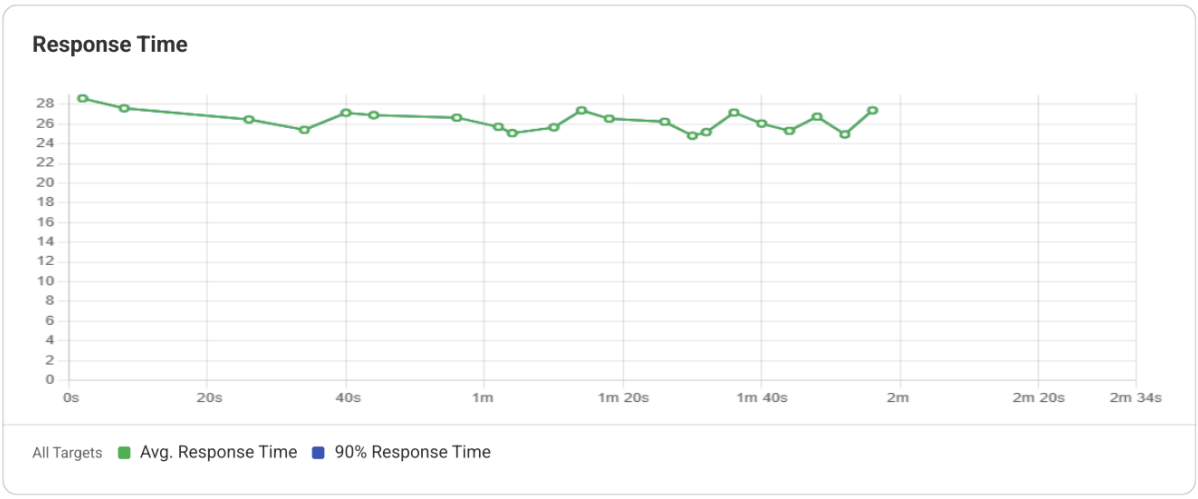


Figure 16 Dotcom-Monitor response time graph from the Stress Report

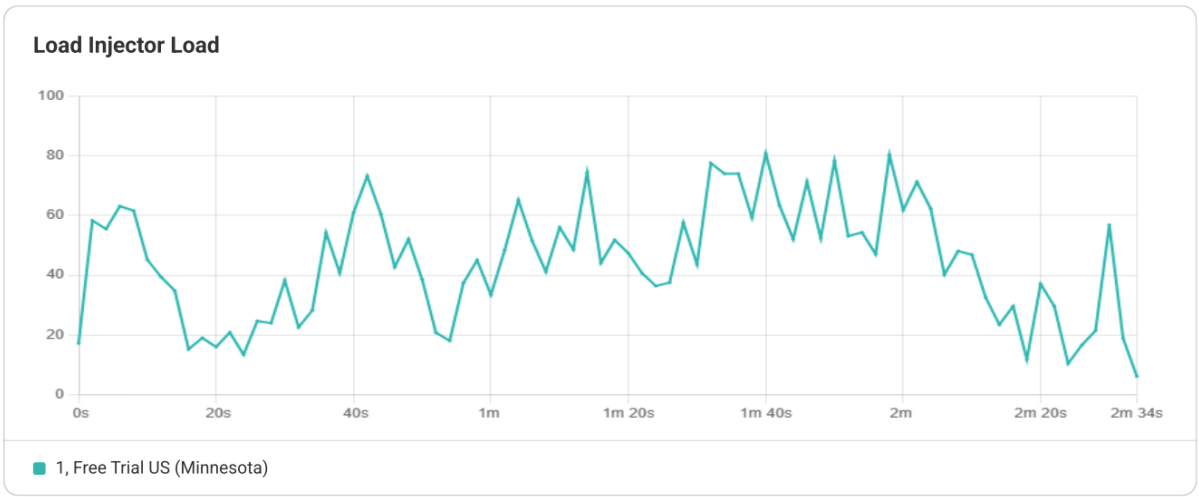


Figure 17 Dotcom-Monitor injector CPU chart

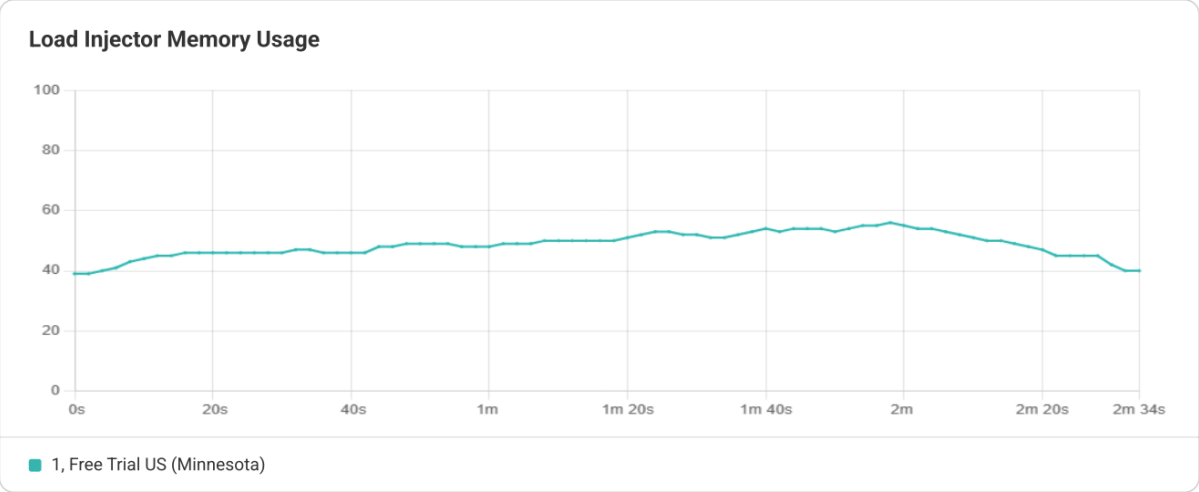


Figure 18 Dotcom-Monitor injector CPU chart

Figure 16, Figure 17 and Figure 18 is given for the graphical representation of the test results from the Dotcom-Monitor

5.4 Reliability and Error Profiles

- Artillery: Backend reliability is great under synthetic load (API → Lambda → DynamoDB/S3) because there are actually no failed responses at any level.
- Dotcom-Monitor: Two runs at 100% success and one with 66% failed sessions indicate mixed reliability. This highlights how end-to-end flows (auth redirection, SPA, network) and external execution context affect user-facing reliability.

5.5 Cost Analysis (AWS Estimated Bill)

5.5.1 Measured Serverless Costs

Service	**Usage Highlight**	**Cost (USD)**	**Notes**
API Gateway (REST)	26,111,854 requests	91.39	Dominant cost driver
Lamda	•	0.00	No charge recorded
DynamoDB	-	0.00	Within free tier

S3 (storage + requests)	0.818 GB-Mo; 1,021 write-like requests	0.03	Negligible
Cognito	1 MAU	0.02	Negligible
ECS/Fargate (ancillary)	78.5 GB-hr; 39.2 vCPU-hr	1.94	Non-critical for app path
VPC public IPv4	19.868 hours	0.10	Minor
Subtotal (pre-tax)	-	93.48	US-East (N.Virginia) highest region spend
Taxes	-	17.75	Mainly on API Gateway & ECS
Estimated Grand Total	-	111.23	Pending bill status

Tabelle 5 AWS cost analysis

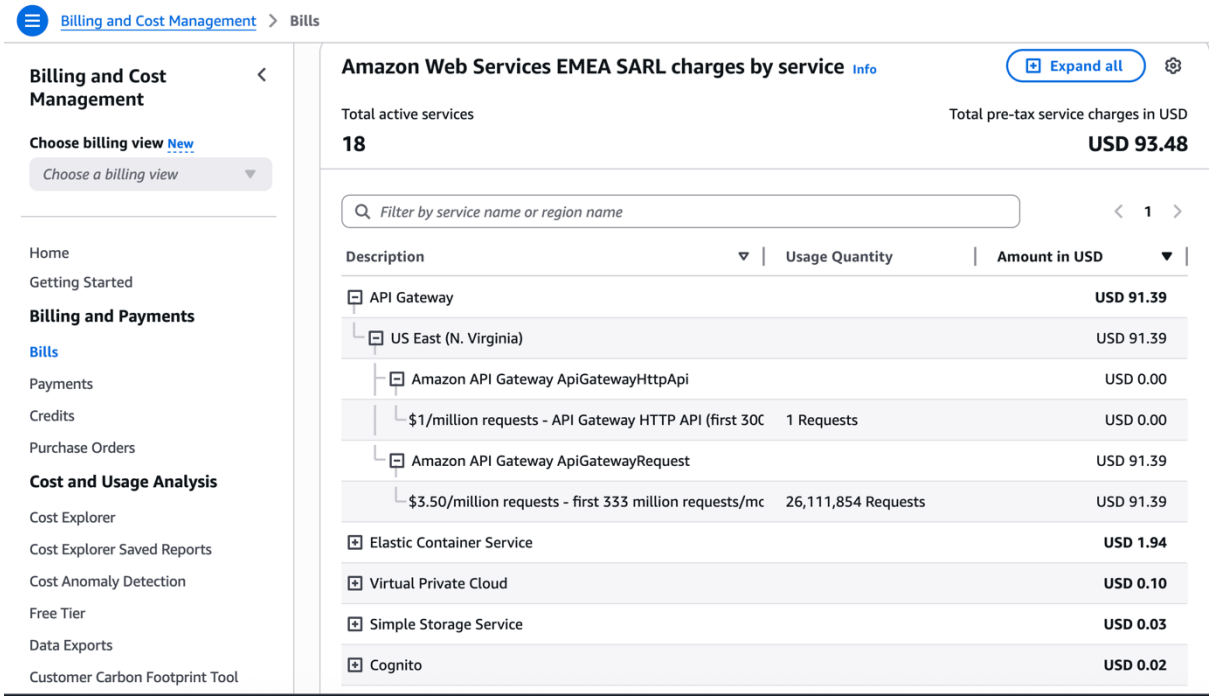


Figure 19 AWS Cost and Usage Analysis

- Derived unit cost (API Gateway): In accordance with the REST API pricing tier, 26,111,854 requests cost USD 91.39 ⇒ ≈ USD 3.50 per million requests. Ref Tabelle 5 and Figure 19
- Traffic normalisation: 26,111,854 requests in 31 days ≈ an average of about 9.75 requests per sec.

This explains why Lambda and DynamoDB charges were negligible: short synthetic bursts at very high RPS, but **low monthly average load**.

5.5.2 Serverless vs Traditional Comparison

For an average ~10 RPS CRUD web application, the Tabelle 6 below compare the actual serverless spend with realistic EC2-based configurations.

Option	Infra Assumption	Estimated Monthly Cost (USD)	Pros	Cons
Serverless (measured)	API GW + Lambda + DynamoDB + S3 + Cognito	111	Zero ops, auto-scale, fine-grained billing	API GW per-request cost dominates at moderate volume
Single EC2 (t3.small)	1 × t3.small + EBS + basic security	~20–25	Cheaper at ~10 RPS; full control	Manual ops, scaling complexity, single point of failure
HA EC2 behind ALB	2 × t3.small (ASG min=2) + ALB + EBS	~60–80	~60–80	Ops overhead; must size for peak; idle waste
Container on Fargate	1 small service auto-scaling, light usage	~30–60	Easier ops than EC2; pay per vCPU/GB-hr	Still pay for idle; need scaling policies

Tabelle 6 Comparison Traditional and Serverless

Due to the API Gateway's cost per request, a modest EC2 can be more affordable than serverless at low-to-moderate constant traffic (~10 RPS). On the contrary, serverless wins in terms of operational effort, burst management, and zero idle cost. Serverless can frequently result in a reduced total cost of ownership and significantly less operational complexity for erratic traffic or situations where peak > average (such as flash bursts reaching tens of thousands of RPS).

5.6 Consolidated Comparison (Performance & Cost)

Dimension	Artillery (Backend)	Dotcom-Monitor (End-to-end)	Comment
Latency(mean)	68-97 ms	26-73 s	User percieved latency dominated by auth + SPA + network
Tail latency	P95: 69-156 ms; max up to 470 ms	~25 s (p90 in stable runs)	Backend tails grow only under extreme load
Reliability	100% success; 0 errors	100% success (two runs) vs 66% failed (one run)	External variability matters
Scalability	Linear up to ~30k RPS; graceful at ~55k	Limited by injector/network	Confirms Lambda autoscaling
Monthly cost	-	API GW: USD 91.39 of USD 111.23 total	API er-request cost dominates

Tabelle 7 Comparison of Load Testing

Summary

- Backend: Up to around 30k RPS, it is incredibly scalable and dependable with a latency of under 100 ms; below 55k RPS, it predictably degrades.
- User-perceived: Stable runs contain response times of 25–29 s; a failing run suggests external fragility.
- Cost: Spend dominates by API Gateway (USD 91.39). EC2 might be cheaper for monthly averages of about 10 RPS; serverless still prevails for unpredictable and bursty loads and ease of operations. {ref Tabelle 7}

Chapter 6: Discussion

6.1 Interpreting Latency

The results show clear bifurcation:

1) Backend latency (ms) vs user-perceived latency (seconds)

2) The network path, possible cold travel pathways, SPA boot (JS bundle fetch via CloudFront + hydration), and front-door factors (Cognito auth rerouting and token exchange) are all responsible for the gap.
3) Lambda + DynamoDB maintains its speed under constant API load; the end-to-end experience, not computation, is the bottleneck.

6.2 Scalability vs User Experience

Excellent horizontal scaling and the lack of backend errors have been shown by Artillery. Dotcom-Monitor, however, underlines the reality that external reliability could vary significantly, even at low VUs. This is inline with the larger body of research: serverless removes server bottlenecks but leaves the internet intact. Front-end performance, auth flows, and global distribution all impact tail latencies and the user experience.

Actionable design implications:

- Reduce SPA boot by aggressively compressing, caching, lazy-loading routes, and minifying.
- If features permit, choose HTTP API over REST API Gateway because it is cheaper per request.
- Cache at edges: Lambda@Edge or CloudFront functions for preflight responses or JWT checks.
- If worldwide users are vital, take into account regional replication (multi-region API + DynamoDB global tables).
- To cut down on interactive auth flows, investigate token refresh and silent SSO practices.

6.3 Reliability and Failure Modes

The failure of the Dotcom-Monitor run indicates sensitivity to:

- Auth and session: unsuccessful or sluggish logins lead to unsuccessful sessions.
- Synthetic client saturation: test-side limits are indicated by injector CPU >80%.
- Throttling: Results can be impacted by client retry/backoff techniques and staged limitations set by the API Gateway.

Mitigations:

- Carefully set up Lambda reserved concurrency and API Gateway throttling.
- Exponential backoff with jitter on the client side.
- For important pathways, use Provisioned Concurrency or warmers (but only if UX + cost justifies it).

6.4 Limitations

- Long-tail and diurnal insights are limited by short-duration tests (minutes, not days).
- Single-region injectors must be used for end-to-end results; multi-region viewpoints would improve generalisability.
- Representative pricing is used in cost comparison; exact figures differ by region and feature.

6.5 Implications and Recommendations

- Backend serverless is not the bottleneck for CRUD-heavy SPAs; instead, focus on front-end performance and auth flows.
- Reduce per-request API fees and, if safe, use batch calls to lower costs by moving to HTTP APIs.
- To link UX delays to particular services, make an investment in observability (distributed tracing + correlation IDs).
- Think about using CloudFront Functions/Lambda@Edge's edge compute for JWT processing and non-sensitive response caching.

6.6 Future Work

- End-to-end testing across multiple regions (EU, APAC) to measure geographical impacts.
- Controlled tests using RUM (real user monitoring) beacons to distinguish API latency, SPA boot time, and cognito latency.
- To map a more comprehensive cost–latency frontier, cost experiments were carried out using the HTTP API, GraphQL/AppSync, and Provisioned Concurrency.

References

Amareswer, 2025. Automate Serverless Deployments with GitHub Actions. Medium. URL https://medium.com/@amareswer/automate-serverless-deployments-with-github-actions-c8a95d7a054a (accessed 8.3.25).

Amazon Congnito, n.d. Using role-based access control - Amazon Cognito [WWW Document]. URL https://docs.aws.amazon.com/cognito/latest/developerguide/role-based-access-control.html?utm_source=chatgpt.com (accessed 8.7.25).

Andersen, G., 2025. Effective CICD Strategies to Reduce Deployment Failures in Startups [WWW Document]. URL https://moldstud.com/articles/p-effective-cicd-strategies-to-reduce-deployment-failures-in-startups (accessed 8.3.25).

Anjali, Caraza-Harter, T., Swift, M., 2020. Blending Containers and Virtual Machines: A Study of Firecracker and gVisor (VEE 2020) - VEE 2020 [WWW Document]. URL https://conf.researchr.org/details/vee-2020/vee-2020-papers/1/Blending-Containers-and-Virtual-Machines-A-Study-of-Firecracker-and-gVisor?utm_source=chatgpt.com (accessed 8.10.25).

Anselmi, J., Gaujal, B., Rebuffi, L.-S., 2025. Non-Stationary Gradient Descent for Optimal Auto-Scaling in Serverless Platforms. IEEE Trans. Netw. 1–14. https://doi.org/10.1109/TON.2025.3538982

Aslam, A., 2025. How We Slashed AWS Lambda Cold Starts by 90% with Node.js [WWW Document]. DEV Community. URL https://dev.to/alex_aslam/how-we-slashed-aws-lambda-cold-starts-by-90-with-nodejs-1489 (accessed 8.1.25).

Asplund, J.-E., 2022. Vercel, Netlify, and the consumerization of developer tools | Sacra [WWW Document]. URL https://www.sacra.com/research/vercel-netlify-consumerization-dev-tools/?utm_source=chatgpt.com (accessed 8.3.25).

AWS, 2025a. AWS Lambda Pricing [WWW Document]. Amazon Web Services, Inc. URL https://aws.amazon.com/lambda/pricing/ (accessed 7.28.25).

AWS, 2025b. Understanding Lambda function scaling - AWS Lambda [WWW Document]. URL https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html?utm_source=chatgpt.com (accessed 7.29.25).

AWS, 2025c. Challenges when testing serverless applications - AWS Prescriptive Guidance [WWW Document]. URL https://docs.aws.amazon.com/prescriptive-guidance/latest/serverless-application-testing/challenges.html?utm_source=chatgpt.com (accessed 8.6.25).

AWS, 2025d. Test AWS infrastructure by using LocalStack and Terraform Tests - AWS Prescriptive Guidance [WWW Document]. URL https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/test-aws-infra-localstack-terraform.html?utm_source=chatgpt.com (accessed 8.6.25).

AWS, 2024. Lambda quotas - AWS Lambda [WWW Document]. URL https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html (accessed 7.27.25).

AWS Blogs, 2022. Securely retrieving secrets with AWS Lambda | AWS Compute Blog [WWW Document]. URL https://aws.amazon.com/blogs/compute/securely-retrieving-secrets-with-aws-lambda/ (accessed 8.7.25).

AWS Lambda, 2025a. AWS Lambda. Wikipedia.

AWS Lambda, 2025b. Securing Lambda environment variables - AWS Lambda [WWW Document]. URL https://docs.aws.amazon.com/lambda/latest/dg/configuration-envvars-encryption.html?utm_source=chatgpt.com (accessed 8.7.25).

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., Suter, P., 2017. Serverless Computing: Current Trends and Open Problems. https://doi.org/10.48550/ARXIV.1706.03178

Bandara, 2024. Unleashing the Power of DynamoDB: A Deep Dive into Serverless Databases [WWW Document]. DEV Community. URL https://dev.to/virajlakshitha/unleashing-the-power-of-dynamodb-a-deep-dive-into-serverless-databases-130n (accessed 7.29.25).

Barringhaus, J., 2025. 4 AWS Serverless Security Traps & How to Fix Them. Tamnoon. URL https://tamnoon.io/blog/4-aws-serverless-security-traps-in-2025-and-how-to-fix-them/ (accessed 8.7.25).

Beswick, 2021. Operating Lambda: Performance optimization – Part 2 | AWS Compute Blog [WWW Document]. URL https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-2/ (accessed 8.1.25).

Beswick, J., 2024. Comparing design approaches for building serverless microservices | AWS Compute Blog [WWW Document]. URL https://aws.amazon.com/blogs/compute/comparing-design-approaches-for-building-serverless-microservices/ (accessed 8.3.25).

Beswick, J., 2021. Operating Lambda: Performance optimization – Part 1 | AWS Compute Blog [WWW Document]. URL https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/ (accessed 7.29.25).

Beswick, J., 2020. Best practices for organizing larger serverless applications | AWS Compute Blog [WWW Document]. URL https://aws.amazon.com/blogs/compute/best-practices-for-organizing-larger-serverless-applications/ (accessed 8.3.25).

Bilal, M., Canini, M., Fonseca, R., Rodrigues, R., 2021. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions. https://doi.org/10.48550/arXiv.2105.14845

Bilgic, E., 2022. Fastest Runtime For AWS Lambda Functions. Security Boulevard. URL https://securityboulevard.com/2022/03/fastest-runtime-for-aws-lambda-functions/ (accessed 8.1.25).

Biørn-Hansen, A., Grønli, T.-M., Ghinea, G., Alouneh, S., 2019. An Empirical Study of Cross-Platform Mobile Development in Industry. Wireless Communications and Mobile Computing 2019, 1–12. https://doi.org/10.1155/2019/5743892

Blessing, M., 2025. The Rise of Serverless Computing in Enterprise Applications.

Boa, G., 2025. Emulating the Cloud: why you should use Firebase Emulator Suite [WWW Document]. DEV Community. URL https://dev.to/this-is-learning/emulating-the-cloud-why-you-should-use-firebase-emulator-suite-1o42 (accessed 8.6.25).

Borges, M.C., Werner, S., Kilic, A., 2021. FaaSter Troubleshooting -- Evaluating Distributed Tracing Approaches for Serverless Applications, in: 2021 IEEE International Conference on Cloud Engineering (IC2E). pp. 83–90. https://doi.org/10.1109/IC2E52221.2021.00022

Castro, P., Ishakian, V., Muthusamy, V., Slominski, A., 2019. The rise of serverless computing. Commun. ACM 62, 44–54. https://doi.org/10.1145/3368454

Cesar, I., 2021. Serverless is Ready, Developers Are Not [WWW Document]. DEV Community. URL https://dev.to/aws-builders/serverless-is-ready-developers-are-not-12f9 (accessed 8.14.25).

Chaudhary, H., 2020. The serverless development experience. The Innovation. URL https://harshchau.medium.com/the-serverless-development-experience-ed49b81eda1b (accessed 8.14.25).

Chen, W., Miller, S., Gomez, R., Tanaka, A., Watkins, J., 2021. Automated Testing Strategies for Serverless Architectures.

Chintala, S., Narani, S., Ayyalasomayajula, M.M.T., 2018. Exploring Serverless Security: Identifying Security Risks and Implementing Best Practices 10, 588–604.

Cloudthat, 2025. 95% Reduction in Deployment Errors with AWS CodePipeline Approval Gates and Real-Time Notifications. CloudThat Resources. URL https://www.cloudthat.com/resources/case-study/95-reduction-in-deployment-errors-with-aws-codepipeline-approval-gates-and-real-time-notifications/ (accessed 8.3.25).

Crudu, V., 2024. Serverless Governance Implementing policies and controls for compliance [WWW Document]. URL https://moldstud.com/articles/p-serverless-governance-implementing-policies-and-controls-for-compliance (accessed 8.13.25).

Cui, Y., 2020. You are wrong about serverless vendor lock-in. Lumigo. URL https://medium.com/lumigo/you-are-wrong-about-serverless-vendor-lock-in-269685d3ad9b (accessed 8.3.25).

Cui, Y., 2017. aws lambda - compare coldstart time with different languages, memory and code sizes. theburningmonk.com. URL https://theburningmonk.com/2017/06/aws-lambda-compare-coldstart-time-with-different-languages-memory-and-code-sizes/ (accessed 7.29.25).

CyberSapiens, 2025. Privilege Escalation via Overly Permissive IAM Roles - CyberSapiens [WWW Document]. URL https://cybersapiens.com.au/case-study/privilege-escalation-via-overly-permissive-iam-roles/?utm_source=chatgpt.com (accessed 8.7.25).

Datadog, 2025. Trace Context Propagation [WWW Document]. Datadog Infrastructure and Application Monitoring. URL https://docs.datadoghq.com/tracing/trace_collection/trace_context_propagation/ (accessed 8.16.25).

DEV Community, 2025. Managing Secrets in an AWS Serverless Application [WWW Document]. DEV Community. URL https://dev.to/aws-builders/managing-secrets-in-an-aws-serverless-application-2752 (accessed 8.7.25).

Douglas, C., 2019. Keeping the Security and Scalability of Serverless Apps Problem-Free with AWS Secrets Manager | AWS Partner Network (APN) Blog [WWW Document]. URL https://aws.amazon.com/blogs/apn/keeping-the-security-and-scalability-of-serverless-apps-problem-free-with-aws-secrets-manager/ (accessed 8.7.25).

Drasner, S., 2021. Developer Experience at Netlify [WWW Document]. URL https://www.netlify.com/blog/2021/01/06/developer-experience-at-netlify/ (accessed 8.14.25).

Durbin, J., 2017. Evaluating API Gateway as a Proxy to internal AWS resources via Lambda and HTTP Proxy [WWW Document]. URL https://www.joshdurbin.net/posts/2017-04-initial-performance-benchmarks-python-api-gateway-proxy-vs-elb/ (accessed 8.3.25).

Ebrahimi, A., Ghobaei-Arani, M., Saboohi, H., 2024. Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions. Journal of Systems Architecture 151, 103115. https://doi.org/10.1016/j.sysarc.2024.103115

Eismann, S., Herbst, N., Kounev, S., 2020. Sizeless: Predicting the optimal size of serverless functions, in: Middleware ’20: ACM/IFIP International Middleware Conference.

Eismann, S., Scheuner, J., Eyk, E.V., Schwinger, M., Grohmann, J., Herbst, N., Abad, C.L., Iosup, A., 2022. The State of Serverless Applications: Collection, Characterization, and Community Consensus. IIEEE Trans. Software Eng. 48, 4152–4166. https://doi.org/10.1109/TSE.2021.3113940

Eivy, A., Weinman, J., 2020. Be Wary of the Economics of “Serverless” Cloud Computing. IEEE Cloud Computing.

Eivy, A., Weinman, J., 2017. Be Wary of the Economics of “Serverless” Cloud Computing. IEEE Cloud Computing 4, 6–12. https://doi.org/10.1109/MCC.2017.32

Fan, C.-F., Jindal, A., Gerndt, M., 2020. Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application, in: Proceedings of the 10th International Conference on Cloud Computing and Services Science. Presented at the 10th International Conference on Cloud Computing and Services Science, SCITEPRESS - Science and Technology Publications, Prague, Czech Republic, pp. 204–215. https://doi.org/10.5220/0009792702040215

Filichkin, A., 2021. AWS Lambda battle 2021: performance comparison for all languages (cold and warm start) [WWW Document]. Medium. URL https://filia-aleks.medium.com/aws-lambda-battle-2021-performance-comparison-for-all-languages-c1b441005fd1 (accessed 8.1.25).

Fred, T., Olasehinde, T., 2019. Monitoring and Observability in Serverless Environments.

Gao, H., 2024. Optimization Research and Solutions for the Cold Start Problem in Serverless Computing. HSET 85, 113–120. https://doi.org/10.54097/shdwt693

Ghorbian, M., Ghobaei-Arani, M., 2025. Serverless Computing: Architecture, Concepts, and Applications. https://doi.org/10.48550/arXiv.2501.09831

Ghosheh, E., Qaddour, J., Kuofie, M., Black, S., 2006. A Comparative Analysis of Maintainability Approaches for Web Applications. pp. 1155–1158. https://doi.org/10.1109/AICCSA.2006.205235

Gomes, Y., 2021. Optimising Serverless Cold Starts. Cazoo Technology Blog. URL https://medium.com/cazoo/optimising-serverless-cold-starts-d199da824f08 (accessed 8.1.25).

Gottlieb, N., 2016. State of the Serverless Community Survey Results [WWW Document]. URL https://www.serverless.com/blog/state-of-serverless-community (accessed 8.3.25).

Greiler, M., Storey, M.-A., Noda, A., 2023. An Actionable Framework for Understanding and Improving Developer Experience. IEEE Transactions on Software Engineering 49, 1411–1425. https://doi.org/10.1109/TSE.2022.3175660

Hamza, M., Akbar, M.A., Capilla, R., 2023. Understanding Cost Dynamics of Serverless Computing: An Empirical Study. https://doi.org/10.48550/arXiv.2311.13242

Hefnawy, E., 2016. Serverless Code Patterns [WWW Document]. URL https://www.serverless.com/blog/serverless-architecture-code-patterns?utm_source=chatgpt.com (accessed 8.3.25).

Helendi, A., 2019. Serverless Survey: +77% Delivery Speed, 4 Dev Workdays/Mo Saved & -26% AWS Monthly Bill. HackerNoon.com. URL https://medium.com/hackernoon/serverless-survey-77-delivery-speed-4-dev-workdays-mo-saved-26-aws-monthly-bill-d99174f70663 (accessed 8.3.25).

Hellerstein, J.M., Faleiro, J., Gonzalez, J.E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C., 2018. Serverless Computing: One Step Forward, Two Steps Back. https://doi.org/10.48550/ARXIV.1812.03651

Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., 2016. Serverless Computation with OpenLambda.

Hevner, A., Chatterjee, S., 2010. Design Research in Information Systems: Theory and Practice. Springer Science & Business Media.

Hui, X., Xu, Y., Shen, X., 2025. Exploring Function Granularity for Serverless Machine Learning Application with GPU Sharing. NaN 1–28. https://doi.org/10.1145/3711699

Implementing IAM in Serverless Architectures: Combining Managed Identity Services with Fine-Grained Authorization for Secure FaaS Across Major Cloud Providers | Volito, 2025. URL https://volito.digital/implementing-iam-in-serverless-architectures-combining-managed-identity-services-with-fine-grained-authorization-for-secure-faas-across-major-cloud-providers/ (accessed 8.3.25).

Jain, A., 2025. Serverless Computing: A Comprehensive Survey. https://doi.org/10.2139/ssrn.5099133

Jin, R., Cordingly, R., Zhao, D., Lloyd, W., 2024. GraphQL vs. REST: A Performance and Cost Investigation for Serverless Applications, in: Proceedings of the 10th International Workshop on Serverless Computing, WoSC10 '24. Association for Computing Machinery, New York, NY, USA, pp. 37–42. https://doi.org/10.1145/3702634.3702956

Joel, A., 2023. SECURITY CHALLENGES AND BEST PRACTICES IN SERVERLESS DATA ENGINEERING.

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A., 2019a. Cloud Programming Simplified: A Berkeley View on Serverless Computing. https://doi.org/10.48550/arXiv.1902.03383

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A., 2019b. Cloud Programming Simplified: A Berkeley View on Serverless Computing. https://doi.org/10.48550/arXiv.1902.03383

Joosen, A., Hassan, A., Asenov, M., Singh, R., Darlow, L., Wang, J., Deng, Q., Barker, A., 2025. Serverless Cold Starts and Where to Find Them, in: Proceedings of the Twentieth European Conference on Computer Systems. Presented at the EuroSys '25: Twentieth European Conference on Computer Systems, ACM, Rotterdam Netherlands, pp. 938–953. https://doi.org/10.1145/3689031.3696073

Kaffes, K., Yadwadkar, N.J., Kozyrakis, C., 2019. Centralized Core-granular Scheduling for Serverless Functions, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC '19. Association for Computing Machinery, New York, NY, USA, pp. 158–164. https://doi.org/10.1145/3357223.3362709

Khatri, D., Khatri, S.K., Mishra, D., 2020. Potential Bottleneck and Measuring Performance of Serverless Computing: A Literature Study, in: 2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO). Presented at the 2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), pp. 161–164. https://doi.org/10.1109/ICRITO48877.2020.9197837

Kunz, I., Casola, V., Schneider, A., Banse, C., Schütte, J., 2020. Towards Tracking Data Flows in Cloud Architectures, in: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). pp. 445–452. https://doi.org/10.1109/CLOUD49709.2020.00066

Lapin, S., 2021. Lessons learned on concurrency limits of AWS Lambda backed by RDS database [WWW Document]. Sergey Lapin. URL https://svlapin.github.io/engineering/2021/04/05/aws-lambda-rds-scaling.html (accessed 7.29.25).

Lee, C., Zhu, Z., Yang, T., Huo, Y., Su, Y., He, P., Lyu, M.R., 2024. SPES: Towards Optimizing Performance-Resource Trade-Off for Serverless Functions. https://doi.org/10.48550/arXiv.2403.17574

Leffler, G., 2021. How to Use Observability to Reduce MTTR | Splunk [WWW Document]. URL https://www.splunk.com/en_us/blog/devops/using-observability-to-reduce-mttr.html?utm_source=chatgpt.com (accessed 8.4.25).

Leiß, S., 2025. Designing a Scalable SPA with AWS Cognito and Advanced Access Control. Medium. URL https://medium.com/@svenleiss/designing-a-scalable-spa-with-aws-cognito-and-advanced-access-control-81f3328451c8 (accessed 8.7.25).

Leitner, P., Wittern, E., Spillner, J., Hummer, W., 2019. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. Journal of Systems and Software 149, 340–359. https://doi.org/10.1016/j.jss.2018.12.013

Lewis, J., 2021. Tell me your secrets: Serverless Secrets in AWS Lambda. Orca Security. URL https://orca.security/resources/blog/aws-lambda-secrets/ (accessed 8.7.25).

Li, X., Leng, X., Chen, Y., 2021. Securing Serverless Computing: Challenges, Solutions, and Opportunities. https://doi.org/10.48550/arXiv.2105.12581

Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., Pallickara, S., 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance, in: 2018 IEEE International Conference on Cloud Engineering (IC2E). Presented at the 2018 IEEE International Conference on Cloud Engineering (IC2E), pp. 159–169. https://doi.org/10.1109/IC2E.2018.00039

Lüdemann, C., 2020. Setting Up A Multi-Tenant Application With Firebase, GraphQL, and Angular. Christian Lüdemann. URL https://christianlydemann.com/setting-up-a-multi-tenant-application-with-firebase-graphql-and-angular/ (accessed 8.10.25).

Magee, D., 2025. The Overlooked Six | AWS Security Blind Spots. SentinelOne. URL https://www.sentinelone.com/blog/the-overlooked-six-aws-security-blind-spots/ (accessed 8.7.25).

Marcelino, C., Gollhofer-Berger, S., Pusztai, T., Nastic, S., 2025. Cosmos: A Cost Model for Serverless Workflows in the 3D Compute Continuum. https://doi.org/10.48550/arXiv.2504.20189

Mărcuță, C., 2024. Optimizing Performance with PaaS - A Guide to Efficient App Development [WWW Document]. URL https://moldstud.com/articles/p-optimizing-performance-with-paas-app-development (accessed 8.16.25).

Marin, E., Diego, P., Pietro, R.D., 2022a. Serverless computing: a security perspective | Journal of Cloud Computing [WWW Document]. URL https://link.springer.com/article/10.1186/s13677-022-00347-w (accessed 8.3.25).

Marin, E., Perino, D., Pietro, R.D., 2022b. Serverless Computing: A Security Perspective. https://doi.org/10.48550/arXiv.2107.03832

Markovic, D., Scekic, M., Bucaioni, A., Cicchetti, A., 2022. Could jamstack be the future of web applications architecture?: an empirical study | Request PDF [WWW Document]. URL https://www.researchgate.net/publication/360442603_Could_jamstack_be_the_future_of_web_applications_architecture_an_empirical_study (accessed 7.24.25).

Marquez, E., 2023. Evaluate serverless computing best practices | TechTarget [WWW Document]. Search Cloud Computing. URL https://www.techtarget.com/searchcloudcomputing/tip/Evaluate-serverless-computing-best-practices (accessed 8.3.25).

McGrath, G., Brenner, P.R., 2017. Serverless Computing: Design, Implementation, and Performance, in: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). Presented at the 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), IEEE, Atlanta, GA, USA, pp. 405–410. https://doi.org/10.1109/ICDCSW.2017.36

Microservices: Yesterday, Today, and Tomorrow, 2017. , in: Present and Ulterior Software Engineering. Springer International Publishing, Cham, pp. 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

Microsoft Learn, 2025. Azure Functions best practices [WWW Document]. URL https://learn.microsoft.com/en-us/azure/azure-functions/functions-best-practices (accessed 7.29.25).

Mitchell, A., 2024. The Best Backend As A Service For Your React App: A 2023 Comparison Guide - ExpertBeacon [WWW Document]. URL https://expertbeacon.com/the-best-backend-as-a-service-for-your-react-app-a-2023-comparison-guide/ (accessed 8.3.25).

Moebius, F., Pfandzelter, T., Bermbach, D., 2024. Are Unikernels Ready for Serverless on the Edge.

Netlify, 2025a. Netlify. Wikipedia.

Netlify, 2025b. Functions overview [WWW Document]. Netlify Docs. URL https://docs.netlify.com/build/functions/overview/ (accessed 8.16.25).

Newman, S., 2019. Monolith to Microservices.

Nguyen, X., 2024. How serverless computing revolutionizes software development [WWW Document]. Future of software. URL https://www.future-of-software.com/blog/how-serverless-computing-revolutionizes-software-development (accessed 8.14.25).

Oliveira, A., 2020. Backend For Frontend: A tailored strategy for delivering microservices. Medium. URL https://allan-oliveira.medium.com/backend-for-frontend-a-tailored-strategy-for-delivering-microservices-5736a3806b6c (accessed 8.3.25).

OWASP, 2025. Authentication - OWASP Cheat Sheet Series [WWW Document]. URL https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html (accessed 8.3.25).

Paghar, F., 2024. React Component Design Patterns - Part 1 [WWW Document]. DEV Community. URL https://dev.to/fpaghar/react-component-design-patterns-part-1-5f0g (accessed 8.14.25).

Pahl, C., Brogi, A., Soldani, J., Jamshidi, P., 2019. Cloud Container Technologies: A State-of-the-Art Review. IEEE Transactions on Cloud Computing 7, 677–692. https://doi.org/10.1109/TCC.2017.2702586

Pandey, S., Barker, M., 2025. AWS Lambda introduces tiered pricing for Amazon CloudWatch logs and additional logging destinations | AWS Compute Blog [WWW Document]. URL https://aws.amazon.com/blogs/compute/aws-lambda-introduces-tiered-pricing-for-amazon-cloudwatch-logs-and-additional-logging-destinations/ (accessed 7.28.25).

Plößer, K., 2021. AWS Lambda Monitoring: The Best Tools | Dashbird [WWW Document]. URL https://dashbird.io/blog/top-aws-lambda-performance-monitoring-tools/?utm_source=chatgpt.com (accessed 8.16.25).

Prasad, V., 2021. Scaling AWS Lambda to 30k Requests Per Second | Serverless Guru [WWW Document]. URL https://www.serverlessguru.com/blog/scaling-aws-lambda-to-30k-request-per-second, https://www.serverlessguru.com/blog/scaling-aws-lambda-to-30k-request-per-second (accessed 7.29.25).

Reddit, 2024a. Using Firebase Cloud Functions as a BFF for Firestore optimization [WWW Document]. URL https://www.reddit.com/r/Firebase/comments/18xszan/firebase_a_nobrainer_or_not/?utm_source=chatgpt.com (accessed 8.3.25).

Reddit, 2024b. I am replacing my app with a Rest API on firebase functions : r/Firebase [WWW Document]. URL

https://www.reddit.com/r/Firebase/comments/1drxx5h/i_am_replacing_my_app_with_a_rest_api_on_firebase/?utm_source=chatgpt.com (accessed 8.3.25).
Richardson, C., 2018. Microservice Patterns: With examples in Java.
Roberts, M., 2018. Serverless Architectures [WWW Document]. martinfowler.com. URL https://martinfowler.com/articles/serverless.html (accessed 8.16.25).
Saroar, S.G., Nayebi, M., 2023. Developers' Perception of GitHub Actions: A Survey Analysis. https://doi.org/10.48550/arXiv.2303.04084
Satzger, B., Hummer, W., Inzinger, C., 2013. Winds of Change: From Vendor Lock-In to the Meta Cloud. Internet Computing, IEEE 17, 69–73. https://doi.org/10.1109/MIC.2013.19
Savants, S., 2025. Top 5 Serverless Platforms for Hosting React Apps. URL https://www.serverlessservants.org/top-5-serverless-platforms-for-hosting-react-applications/ (accessed 8.3.25).
Scheuner, J., n.d. Performance Evaluation of Serverless Applications and Infrastructures.
Serverless Savants, 2025. Serverless Serverless Observability Best Practices – Serverless Savants. URL https://serverlesssavants.org/serverless-serverless-observability-best-practices (accessed 8.4.25).
Shafiei, H., Khonsari, A., Mousavi, P., 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. ACM Comput. Surv. 54, 1–32. https://doi.org/10.1145/3510611
Shah, Z., 2024. A Complete Guide to React Architecture Patterns. Medium. URL https://devshi-bambhaniya.medium.com/a-complete-guide-to-react-architecture-patterns-ea386d2ba327 (accessed 8.14.25).
Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., Bianchini, R., 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. Presented at the 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp. 205–218.
Shastri, S., Wasserman, M., Chidambaram, V., 2019. GDPR Anti-Patterns: How Design and Operation of Modern Cloud-scale Systems Conflict with GDPR. https://doi.org/10.48550/arXiv.1911.00498
Shillaker, S., Pietzuch, P., 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. Presented at the 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp. 419–433.
Shillaker, S., Pietzuch, P., n.d. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing.
Siidorow, M., 2024. Survey of Serverless Edge Computing for Web Applications. https://doi.org/10.13140/RG.2.2.13600.39680
Specmatic, 2023. Case Study: How an Enterprise Team Achieved a 75% Reduction in Cycle Time for APIs - Specmatic [WWW Document]. https://specmatic.io/. URL https://specmatic.io/case-studies/case-study-cdd-cut-api-cycle-time-by-75-percent/ (accessed 8.16.25).
Spillner, J., 2019. Serverless Computing and Cloud Function-based Applications, in: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion. Presented at the UCC '19: IEEE/ACM 12th International Conference on Utility and Cloud Computing, ACM, Auckland New Zealand, pp. 177–178. https://doi.org/10.1145/3368235.3370269
Syeda, N., Shah, H., Singh, R., 2025. Analysis of Cost-Efficiency of Serverless Approaches. arXiv preprint arXiv:2501.00001.

Taibi, D., Spillner, J., Wawruch, K., 2021. Serverless Computing-Where Are We Now, and Where Are We Heading? IEEE Softw. 38, 25–31. https://doi.org/10.1109/MS.2020.3028708

Ulili, S., 2025. Vercel vs Netlify vs AWS Amplify: Front-End Hosting | Better Stack Community [WWW Document]. URL https://betterstack.com/community/guides/scaling-nodejs/vercel-vs-netlify-vs-aws-amplify/ (accessed 8.3.25).

Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uta, A., Iosup, A., 2018. Serverless is More: From PaaS to Present Cloud Computing. IEEE Internet Comput. 22, 8–17. https://doi.org/10.1109/MIC.2018.053681358

Venčkauskas, A., Kukta, D., Grigaliunas, S., Brūzgienė, R., 2023. Enhancing Microservices Security with Token-Based Access Control Method. Sensors 23, 3363. https://doi.org/10.3390/s23063363

Vercel, 2025a. Next.js on Vercel [WWW Document]. URL https://vercel.com/docs/frameworks/full-stack/nextjs (accessed 8.14.25).

Vercel, 2025b. Configuring Memory and CPU for Vercel Functions [WWW Document]. URL https://vercel.com/docs/functions/configuring-functions/memory (accessed 8.14.25).

Vercel, 2025c. Runtimes [WWW Document]. URL https://vercel.com/docs/functions/runtimes (accessed 8.14.25).

Verdet, A., Hamdaqa, M., Silva, L.D., Khomh, F., 2025. Exploring Security Practices in Infrastructure as Code: An Empirical Study. Empir Software Eng 30, 74. https://doi.org/10.1007/s10664-024-10610-0

Villamizar, M., Garces, O., Ochoa, L., 2017. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures | Service Oriented Computing and Applications [WWW Document]. URL https://link.springer.com/article/10.1007/s11761-017-0208-y (accessed 8.17.25).

Voss, L., 2021. First look at the Jamstack Community Survey 2021 Results [WWW Document]. URL https://www.netlify.com/blog/2021/10/06/first-look-announcing-jamstack-community-survey-2021-results/ (accessed 8.14.25).

Wachtel, J., 2022. Vercel Brings Serverless Functions to the Edge. The New Stack. URL https://thenewstack.io/vercel-brings-serverless-functions-to-the-edge/ (accessed 8.3.25).

Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M., 2018. Peeking Behind the Curtains of Serverless Platforms. Presented at the 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 133–146.

Waseem, M., Liang, P., Shahin, M., Salle, A.D., Márquez, G., 2021. Design, Monitoring, and Testing of Microservices Systems: The Practitioners' Perspective. Journal of Systems and Software 182, 111061. https://doi.org/10.1016/j.jss.2021.111061

Wen, J., Chen, Z., Jin, X., Liu, X., 2022. Rise of the Planet of Serverless Computing: A Systematic Review. https://doi.org/10.48550/arXiv.2206.12275

Wen, J., Chen, Z., Zhu, Z., Sarro, F., Liu, Y., Ping, H., Wang, S., 2025. LLM-Based Misconfiguration Detection for AWS Serverless Computing. ACM Trans. Softw. Eng. Methodol. 3745766. https://doi.org/10.1145/3745766

Weyori, B., Tetteh, S.G., 2024. Systematic Review and Analysis of Cost- Saving Mechanisms, Challenges, And Best Practices in A Serverless Computing Environment 45, 3724–3736. https://doi.org/10.52783/tjjpt.v45.i03.7845

Zetas, 2025. Impact of Serverless Architecture on Software Development Costs | Zetaton [WWW Document]. URL https://www.zetaton.com/blogs/the-impact-of-serverless-architecture-on-software-development-costs (accessed 8.3.25).

Zhao, C., Ren, Z., Wu, Y., Ren, Y., Wan, J., Shi, W., 2017. Quantifying the Isolation Characteristics in Container Environments | SpringerLink [WWW Document]. URL https://link.springer.com/chapter/10.1007/978-3-319-68210-5_17?utm_source=chatgpt.com (accessed 8.7.25).

Zhao, H., Benomar, Z., Pfandzelter, T., Georgantas, N., 2022. Supporting Multi-Cloud in Serverless Computing [WWW Document]. arXiv.org. https://doi.org/10.1109/UCC56403.2022.00051

Appendices

Appendix A: App.jsx

```
var app = app || {};

(function () {
    'use strict';

    app.ALL_TODOS = 'all';
    app.ACTIVE_TODOS = 'active';
    app.COMPLETED_TODOS = 'completed';
    var TodoFooter = app.TodoFooter;
    var TodoItem = app.TodoItem;

    var ENTER_KEY = 13;

    var TodoApp = React.createClass({
       getInitialState: function () {
          return {
             nowShowing: app.ALL_TODOS,
             editing: null,
             newTodo: ''
          };
       },

       componentDidMount: function () {
          var setState = this.setState;
          var router = Router({
             '/': setState.bind(this, {nowShowing: app.ALL_TODOS}),
             '/active': setState.bind(this, {nowShowing:
app.ACTIVE_TODOS}),
             '/completed': setState.bind(this, {nowShowing:
app.COMPLETED_TODOS})
          });
          router.init('/');
       },

       handleChange: function (event) {
          this.setState({newTodo: event.target.value});
       },

       handleNewTodoKeyDown: function (event) {
          if (event.keyCode !== ENTER_KEY) {
             return;
          }

          event.preventDefault();

          var val = this.state.newTodo.trim();

          if (val) {
             this.props.model.addTodo(val);
             this.setState({newTodo: ''});
          }
       },

       toggleAll: function (event) {
          var checked = event.target.checked;
          this.props.model.toggleAll(checked);
```

```
},

toggle: function (todoToToggle) {
   this.props.model.toggle(todoToToggle);
},

destroy: function (todo) {
   this.props.model.destroy(todo);
},

edit: function (todo) {
   this.setState({editing: todo.id});
},

save: function (todoToSave, text) {
   this.props.model.save(todoToSave, text);
   this.setState({editing: null});
},

cancel: function () {
   this.setState({editing: null});
},

clearCompleted: function () {
   this.props.model.clearCompleted();
},

render: function () {
   var footer;
   var main;
   var todos = this.props.model.todos;

   var shownTodos = todos.filter(function (todo) {
      switch (this.state.nowShowing) {
      case app.ACTIVE_TODOS:
         return !todo.completed;
      case app.COMPLETED_TODOS:
         return todo.completed;
      default:
         return true;
      }
   }, this);

   var todoItems = shownTodos.map(function (todo) {
      return (
         <TodoItem
            key={todo.id}
            todo={todo}
            onToggle={this.toggle.bind(this, todo)}
            onDestroy={this.destroy.bind(this, todo)}
            onEdit={this.edit.bind(this, todo)}
            editing={this.state.editing === todo.id}
            onSave={this.save.bind(this, todo)}
            onCancel={this.cancel}
         />
      );
   }, this);

   var activeTodoCount = todos.reduce(function (accum, todo) {
      return todo.completed ? accum : accum + 1;
   }, 0);
```

```
        var completedCount = todos.length - activeTodoCount;

        if (activeTodoCount || completedCount) {
          footer =
            <TodoFooter
              count={activeTodoCount}
              completedCount={completedCount}
              nowShowing={this.state.nowShowing}
              onClearCompleted={this.clearCompleted}
            />;
        }

        if (todos.length) {
          main = (
            <section className="main">
              <input
                  className="toggle-all"
                  type="checkbox"
                  onChange={this.toggleAll}
                  checked={activeTodoCount === 0}
              />
              <ul className="todo-list">
                {todoItems}
              </ul>
            </section>
          );
        }

        return (
          <div>
            <header className="header">
              <h1>todos</h1>
              <input
                  className="new-todo"
                  placeholder="What needs to be done?"
                  value={this.state.newTodo}
                  onKeyDown={this.handleNewTodoKeyDown}
                  onChange={this.handleChange}
                  autoFocus={true}
              />
            </header>
            {main}
            {footer}
          </div>
        );
      }
    });

    var model = new app.TodoModel('react-todos');

    function render() {
      React.render(
        <TodoApp model={model}/>,
        document.getElementsByClassName('todoapp')[0]
      );
    }

    model.subscribe(render);
    render();
})();
```

Appendix B: TodoModel.js

```
var app = app || {};

(function () {
    'use strict';

    var Utils = app.Utils;
    app.TodoModel = function (key) {
       var self = this;
       this.key = key;
       this.todos = [];
       this.onChanges = [];

       Utils.load(function(err, data) {
          if (err) {
             console.error(err);
          } else {
             self.todos = data;
             self.inform();
          }
       });
    };

    app.TodoModel.prototype.subscribe = function (onChange) {
       this.onChanges.push(onChange);
    };

    app.TodoModel.prototype.inform = function () {
       Utils.store(this.todos);
       this.onChanges.forEach(function (cb) { cb(); });
    };

    app.TodoModel.prototype.addTodo = function (title) {
       this.todos = this.todos.concat({
          id: Utils.uuid(),
          title: title,
          completed: false
       });

       this.inform();
    };

    app.TodoModel.prototype.toggleAll = function (checked) {

       this.todos = this.todos.map(function (todo) {
          return Utils.extend({}, todo, {completed: checked});
       });

       this.inform();
    };

    app.TodoModel.prototype.toggle = function (todoToToggle) {
       this.todos = this.todos.map(function (todo) {
          return todo !== todoToToggle ?
             todo :
             Utils.extend({}, todo, {completed: !todo.completed});
       });
```

```
        this.inform();
    };

    app.TodoModel.prototype.destroy = function (todo) {
        this.todos = this.todos.filter(function (candidate) {
            return candidate !== todo;
        });

        this.inform();
    };

    app.TodoModel.prototype.save = function (todoToSave, text) {
        this.todos = this.todos.map(function (todo) {
            return todo !== todoToSave ? todo : Utils.extend({}, todo,
{title: text});
        });

        this.inform();
    };

    app.TodoModel.prototype.clearCompleted = function () {
        this.todos = this.todos.filter(function (todo) {
            return !todo.completed;
        });

        this.inform();
    };

})();
```

Appendix C: Util.js

```
var app = app || {};

(function () {
    'use strict';

    app.Utils = {
        uuid: function () {
            /*jshint bitwise:false */
            var i, random;
            var uuid = '';

            for (i = 0; i < 32; i++) {
                random = Math.random() * 16 | 0;
                if (i === 8 || i === 12 || i === 16 || i === 20) {
                    uuid += '-';
                }
                uuid += (i === 12 ? 4 : (i === 16 ? (random & 3 | 8) :
random))
                    .toString(16);
            }

            return uuid;
        },

        pluralize: function (count, word) {
            return count === 1 ? word : word + 's';
```

```
        },

        // Post to the API to store todos
        store: function (data) {
            fetch(BASE_PATH + 'api/todos', { method: 'POST',
                body: JSON.stringify(data)
            }).then(function(response) {
                return response.json();
            }).then(function(json) {
                console.log('saved');
            }).catch(function(err) {
                console.error(err);
            });
        },

        // Fetch todos from the API
        load: function(callback) {
            fetch(BASE_PATH + 'api/todos').then(function(response) {
                return response.json();
            }).then(function(json) {
                return callback(null, json);
            }).catch(function(err) {
                return callback(err);
            });
        },

        extend: function () {
            var newObj = {};
            for (var i = 0; i < arguments.length; i++) {
                var obj = arguments[i];
                for (var key in obj) {
                    if (obj.hasOwnProperty(key)) {
                        newObj[key] = obj[key];
                    }
                }
            }
            return newObj;
        }
    };
})();
```

Appendix D: todoItem.jsx

```
var app = app || {};

(function () {
    'use strict';

    var ESCAPE_KEY = 27;
    var ENTER_KEY = 13;

    app.TodoItem = React.createClass({
        handleSubmit: function (event) {
            var val = this.state.editText.trim();
            if (val) {
                this.props.onSave(val);
                this.setState({editText: val});
            } else {
                this.props.onDestroy();
            }
```

```
        },

        handleEdit: function () {
           this.props.onEdit();
           this.setState({editText: this.props.todo.title});
        },

        handleKeyDown: function (event) {
           if (event.which === ESCAPE_KEY) {
              this.setState({editText: this.props.todo.title});
              this.props.onCancel(event);
           } else if (event.which === ENTER_KEY) {
              this.handleSubmit(event);
           }
        },

        handleChange: function (event) {
           if (this.props.editing) {
              this.setState({editText: event.target.value});
           }
        },

        getInitialState: function () {
           return {editText: this.props.todo.title};
        },


        shouldComponentUpdate: function (nextProps, nextState) {
           return (
              nextProps.todo !== this.props.todo ||
              nextProps.editing !== this.props.editing ||
              nextState.editText !== this.state.editText
           );
        },

        componentDidUpdate: function (prevProps) {
           if (!prevProps.editing && this.props.editing) {
              var node = React.findDOMNode(this.refs.editField);
              node.focus();
              node.setSelectionRange(node.value.length, node.value.length);
           }
        },

        render: function () {
           return (
              <li className={classNames({
                 completed: this.props.todo.completed,
                 editing: this.props.editing
              })}>
                 <div className="view">
                    <input
                       className="toggle"
                       type="checkbox"
                       checked={this.props.todo.completed}
                       onChange={this.props.onToggle}
                    />
                    <label onDoubleClick={this.handleEdit}>
                       {this.props.todo.title}
                    </label>
                    <button className="destroy"
onClick={this.props.onDestroy} />
```

```
                </div>
                <input
                    ref="editField"
                    className="edit"
                    value={this.state.editText}
                    onBlur={this.handleSubmit}
                    onChange={this.handleChange}
                    onKeyDown={this.handleKeyDown}
                />
            </li>
        );
    }
  });
})();
```

Appendix E: Footer.jsx

```
var app = app || {};

(function () {
    'use strict';

    app.TodoFooter = React.createClass({
        render: function () {
            var activeTodoWord = app.Utils.pluralize(this.props.count,
'item');
            var clearButton = null;

            if (this.props.completedCount > 0) {
                clearButton = (
                    <button
                        className="clear-completed"
                        onClick={this.props.onClearCompleted}>
                        Clear completed
                    </button>
                );
            }

            var nowShowing = this.props.nowShowing;
            return (
                <footer className="footer">
                    <span className="todo-count">
                        <strong>{this.props.count}</strong> {activeTodoWord}
left
                    </span>
                    <ul className="filters">
                        <li>
                            <a
                                href="#/"
                                className={classNames({selected: nowShowing ===
app.ALL_TODOS})}>
                                    All
                            </a>
                        </li>
                        {' '}
                        <li>
                            <a
                                href="#/active"
```

```
                            className={classNames({selected: nowShowing ===
app.ACTIVE_TODOS})}>
                                Active
                            </a>
                        </li>
                        {' '}
                        <li>
                            <a
                                href="#/completed"
                                className={classNames({selected: nowShowing ===
app.COMPLETED_TODOS})}>
                                Completed
                            </a>
                        </li>
                    </ul>
                    {clearButton}
                </footer>
            );
        }
    });
})();
```