From 0 to Millions: A Guide to Scaling Your App - Part 3

ALEX XU
MAR 1, 2023 · PAID

In the first two parts of this series, we explored the traditional approach to building and scaling an application. It started with a single server that ran everything, and gradually evolved to a microservice architecture that could support millions of daily active users.

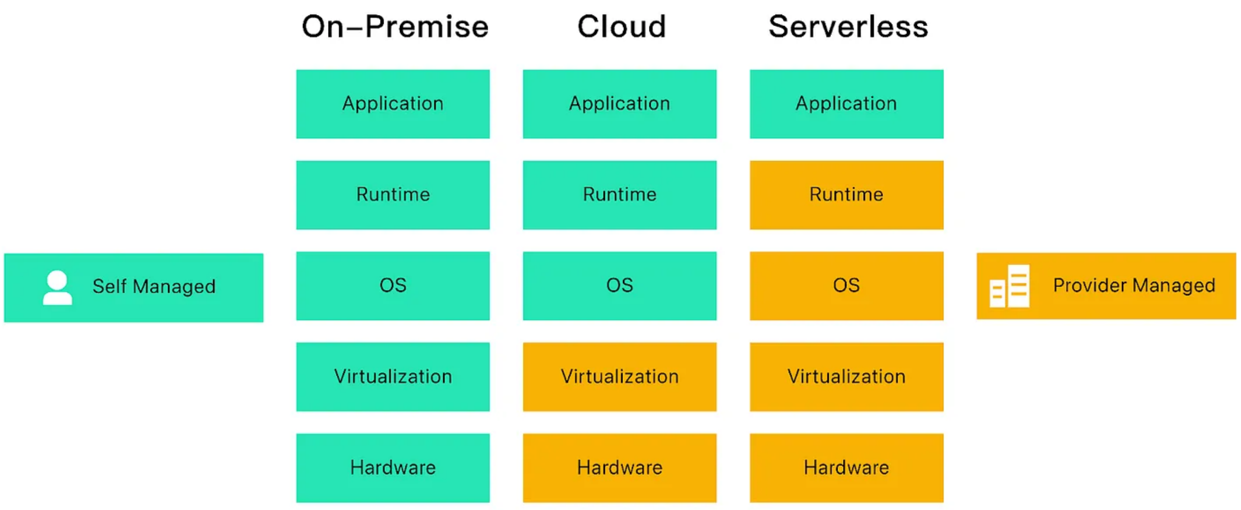
In the final two parts of this series, we examine the impact of recent trends like cloud and serverless computing, along with the proliferation of client application frameworks and the associated developer ecosystem. We explore how these trends alter the way we build applications, especially for early-stage startups where time-to-market is critical, and provide valuable insights on how to incorporate these modern approaches when creating your next big hit.

Recent Trends

Let's start by briefly explaining these computing trends we mentioned.

The first trend is **cloud computing**. Cloud computing, in its most basic form, is running applications on computing resources managed by cloud providers. When using cloud computing, we do not have to purchase or manage hardware ourselves.

The second trend is **serverless computing**. Serverless computing builds on the convenience of cloud computing with even more automation. It enables developers to build and run applications without having to provision cloud servers. The serverless provider handles the infrastructure and automatically scales the computing resources up or down as needed. This provides a great developer experience since developers can focus on the application code itself, without having to worry about scaling.



The third trend rides on the waves of the first two. It is the proliferation of the **client application frameworks and the frontend hosting platforms** that make deploying these frontend applications effortless.

Modern Frontend Frameworks and Hosting Platforms

Let's investigate the third trend a bit more.

In recent years, there has been a significant shift in the way web applications are built. Many of today's popular applications are what is called a Single Page Application (SPA).

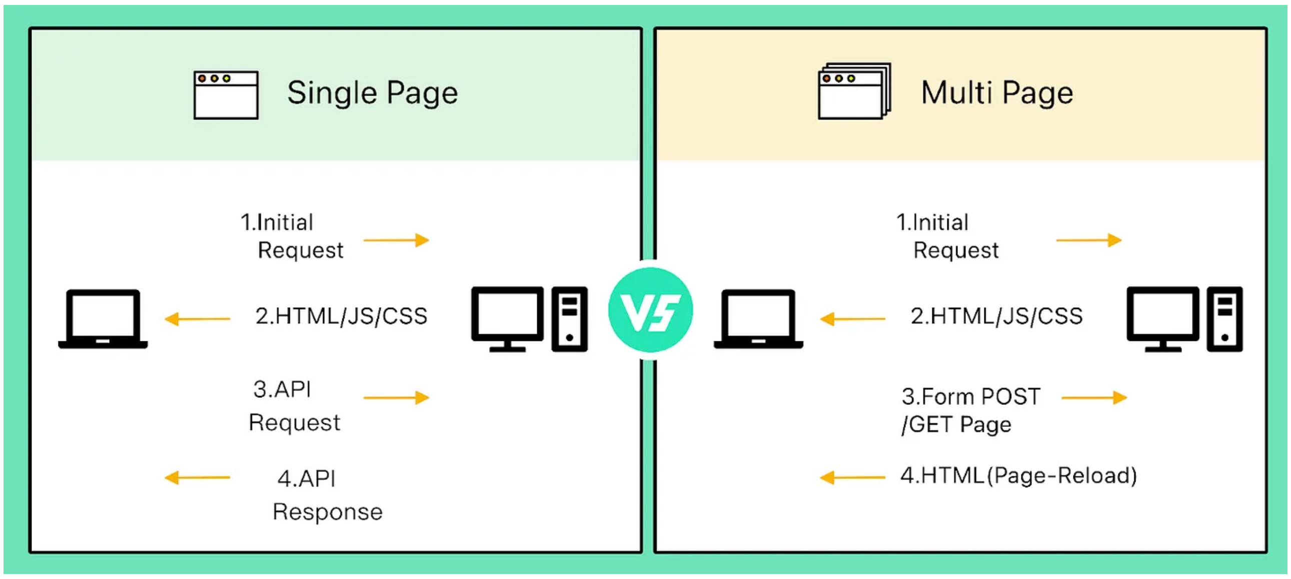
A SPA provides a more seamless user experience by dynamically updating the current page instead of loading a new one each time the user interacts with the application. In a SPA, the initial HTML and its resources are loaded once, and subsequent interactions with the application are handled using JavaScript to manipulate the existing page content.

The traditional way of building web applications, like the one we discussed previously for our e-commerce company Llama, involved serving up new HTML pages from the server every time the user clicked on a link or submitted a form. This conventional model is referred to as a Multi-Page Application (MPA). Each page request typically involves a full page refresh, which could be slow and sometimes disruptive to the user experience.

In contrast, a SPA loads the application's initial HTML frame and then makes requests to the server for data as needed. This approach allows for more efficient use of server resources. The server is not constantly sending full HTML pages and can focus on serving data via a well-defined API instead.

Another benefit of the API-centric approach of a SPA is that the same API is often shared with mobile applications, making the backend easier to maintain.

SPAs are often built using JavaScript frameworks like React. These tools provide a set of abstractions and tools for building complex applications that are optimized for performance and maintainability. In contrast, building an MPA requires a more server-centric approach, which can be more challenging to scale and maintain as the application grows in complexity.

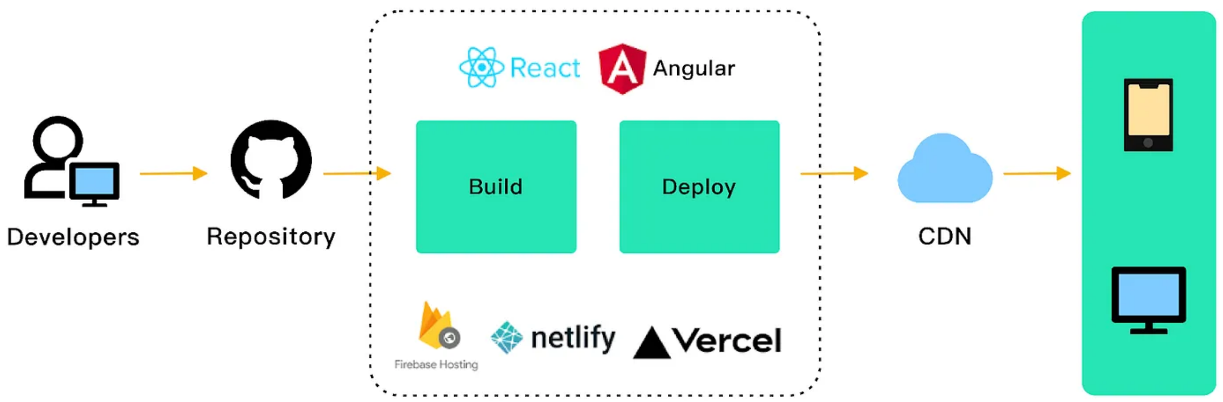


The rise in popularity of these client frameworks brought a wide array of production-grade frontend hosting platforms to the market. Some popular examples include Netify and Vercel. Major cloud providers have similar offerings.

These hosting platforms handle the complexity of building and deploying modern frontend applications at scale. Developers check their code into a repo, and the hosting platforms take over from there. They automatically build the web application bundle and its associated resources and distribute them to the CDN.

Since these hosting platforms are built on the cloud and serverless computing foundation, using best practices like serving data at the edge close to the user, they offer practically infinite scale, and there is no infrastructure to manage.

This is what the modern frontend landscape looks like. The frontend application is built with a modern framework like React. The client application is served by a production-grade hosting platform for scale, and it dynamically fetches data from the backend via a well-defined API.



Modern Backend Options

As mentioned in the previous section, the role of a modern backend is to serve a set of well-defined APIs to support the frontend web and mobile applications.

What are the modern options for building a backend? The shift is similarly dramatic.

Let’s see what a small, resource-constrained startup should use to build its backend, and see how far such a backend could scale.

When time-to-market is critical and resources are limited, a startup should offload as much non-core work as possible. Serverless computing options are attractive, for the following reasons:

- Serverless computing manages the operational aspects of the backend, such as scaling, redundancy, and failover, freeing the startup team from managing infrastructure.
- Serverless computing follows a cost-effective pay-per-use pricing model. There is no up-front commitment.
- Serverless computing allows developers to focus on writing code and testing the backend without worrying about managing servers, leading to shorter time to market.

Serverless Computing Options

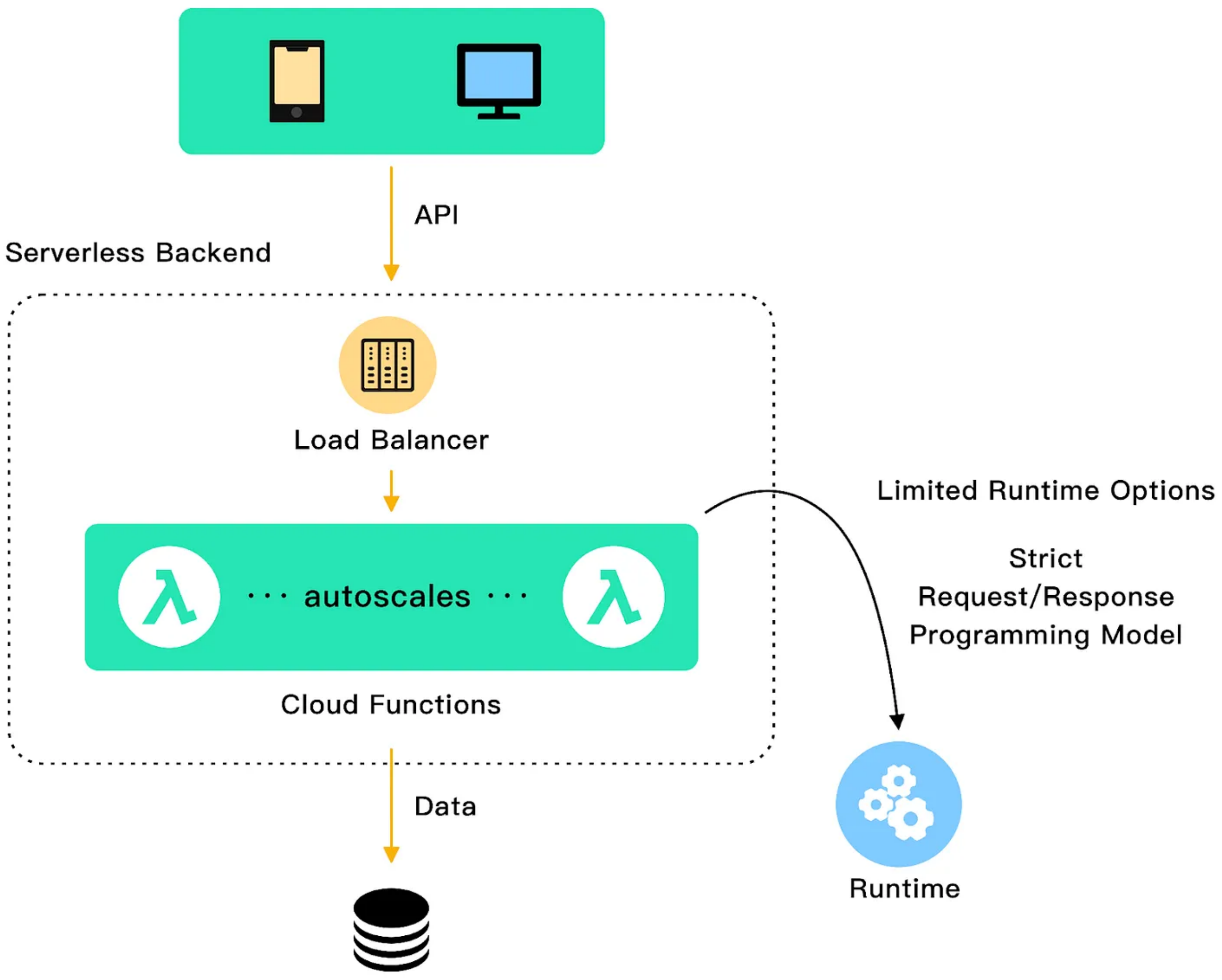
There are two popular types of serverless computing options suitable for a startup - cloud functions like AWS Lambda and Google Cloud Functions, or managed container platforms like Google Cloud Run or AWS App Runner.

Let's explore each one.

Cloud functions are serverless computing platforms designed to handle simple application logic. It is well-suited for applications that require quick and simple execution.

Cloud functions scale automatically based on request volume. They have a simple request/response programming model. However, cloud functions are usually limited to a small number of popular programming languages.

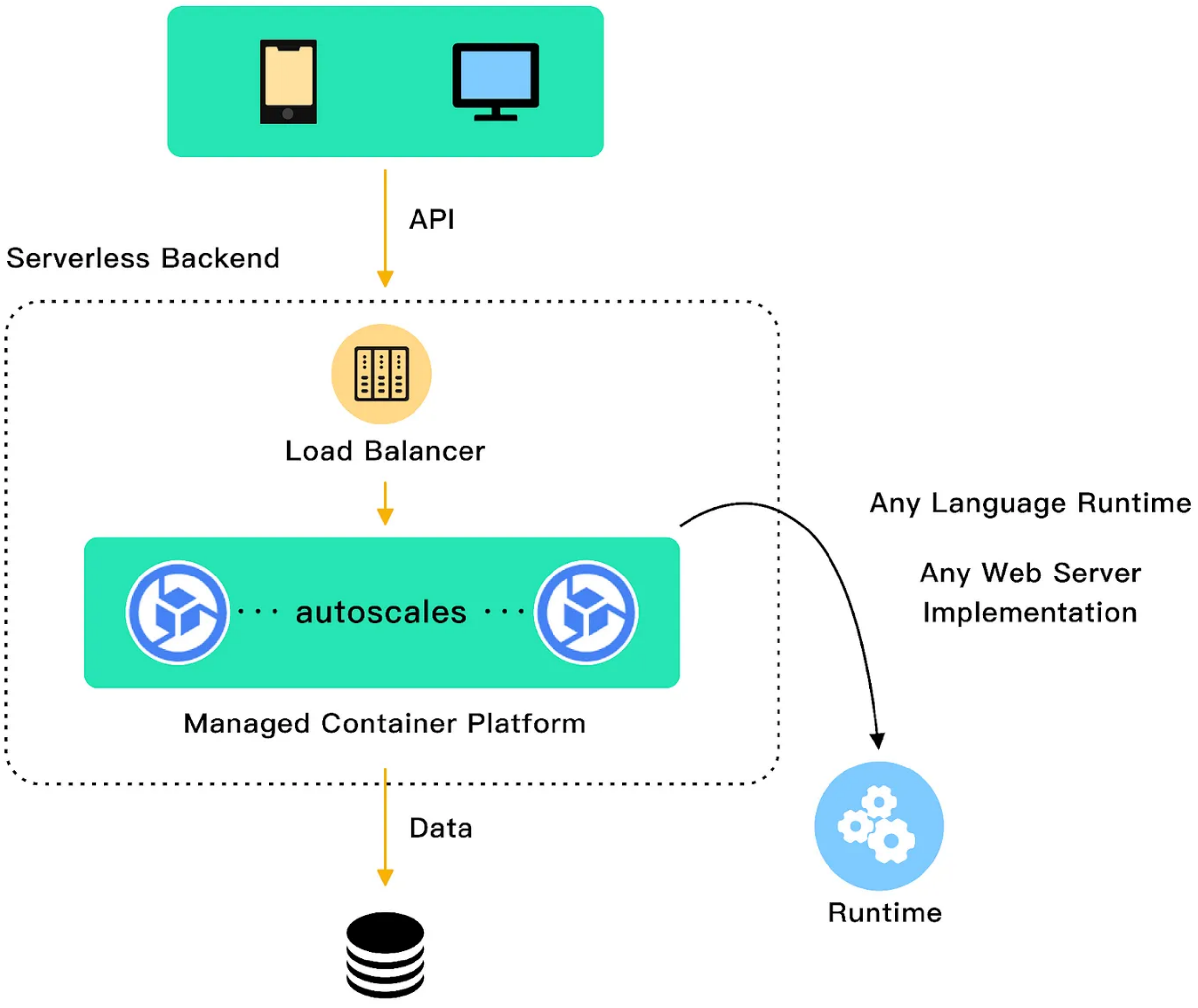
Cloud functions should be able to scale to thousands of simultaneous requests. If the runtime is available for your particular programming environment, cloud functions would be a good choice.



Managed container platforms are similar to cloud functions. The main difference is that these platforms support more runtime environments. We are free to choose most programming environments.

The programming model for a managed container platform is more conventional and flexible. Instead of the simple request/response model of the cloud functions, a managed container platform can run any program inside a container. For an API server workload, each container runs as a typical long-running stateless server. The managed container platform distributes the load across the containers and handles scaling and failover automatically.

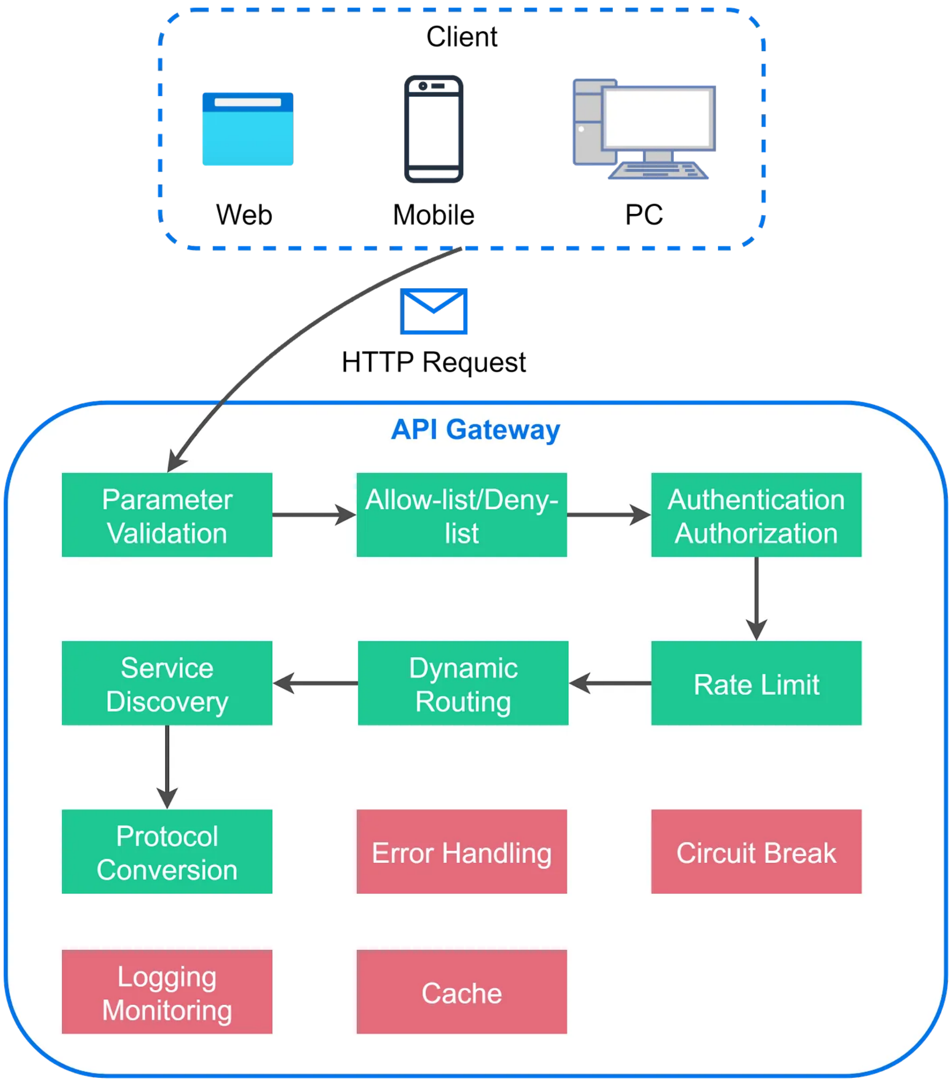
Managed container platforms are a better choice if the application requires more complex runtime environments or a more flexible programming model.



API Gateway

To offload even more basic functionality to the cloud providers, it is a good idea to consider putting an API Gateway in front of the serverless computing options mentioned above.

An API Gateway can be configured to handle authentication and authorization. It provides out-of-the-box monitoring and features like rate limiting and caching, which can help optimize performance and reduce costs.



Serverless Database Options

The database market is also seeing a similarly dramatic shift in recent years. There are a dizzying number of database options available in the market. The challenge for a startup is to pick the right one to start among the many viable options. Choosing a database is a difficult task.

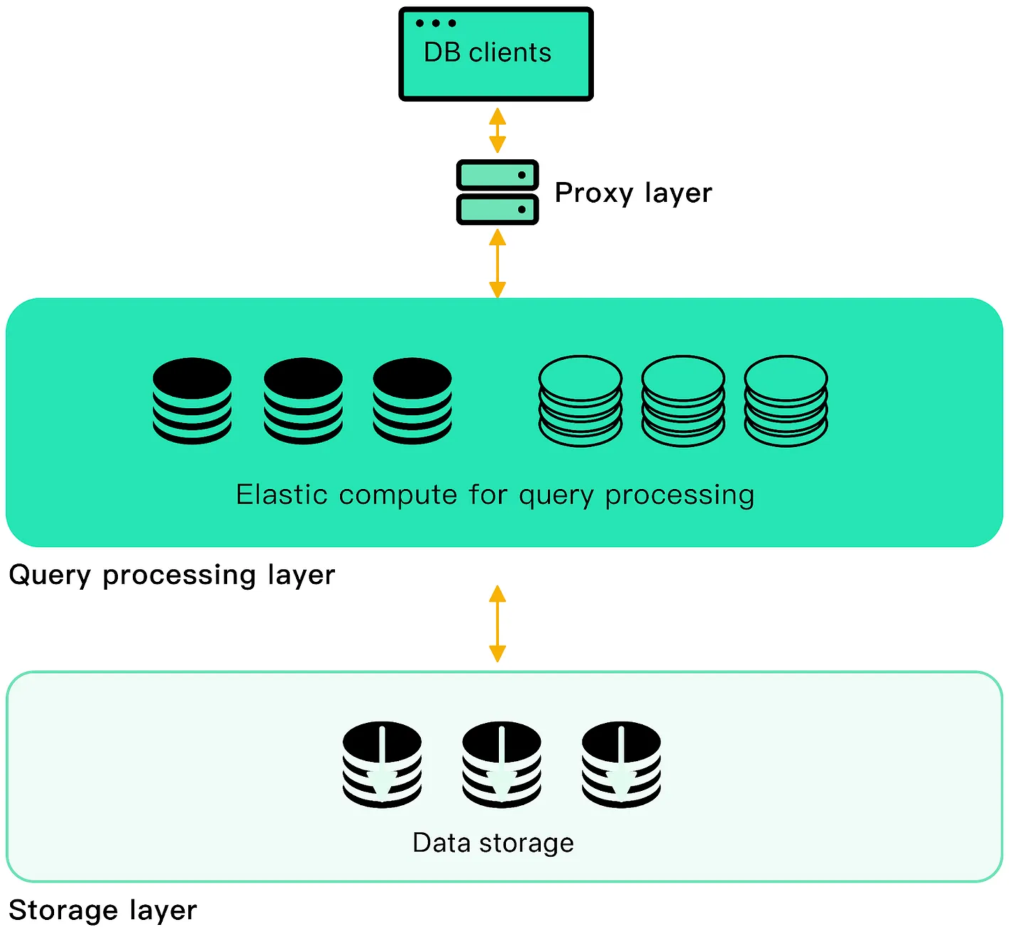
Here is our guidance. Our first piece of advice is to use a relational database. We believe a startup should go with a relational database for several reasons.

First, relational databases are a well-established technology. It offers great schema flexibility that allows a startup to experiment quickly. There is less friction to change course with a relational database compared to NoSQL. With NoSQL, the data schema is more rigid. All the data access patterns must be determined in advance for NoSQL. It greatly limits a startup's ability to pivot.

Second, modern relational databases can run at a high scale. For most API workloads, it is not uncommon for a relational database to handle hundreds of thousands of active users.

Third, when a startup is so successful that it is hitting the limit of a relational database, there is a well-known playbook to scale the database tier, like adding caching tier or introducing read replicas. Refer to part 2 of this series for more information.

There are several ways to operate a production relational database. We would advise choosing a serverless relational database solution. With a serverless database, the cloud provider takes on the majority of the day-to-day operations. A serverless database decouples the data storage from the computing unit that executes queries. The storage and compute units can scale up and down independently based on the amount of traffic, and the customer is only charged for the storage and compute resources that are actually used.



One caution is that serverless databases are relatively new. Some providers have more mature offerings than others. If your particular provider doesn't have a strong offering, a managed database solution is a good alternative. With a managed database, the cloud provider manages the database server hardware. They handle the ongoing maintenance of the underlying hardware and the database software itself. The customer is responsible for some operational tasks, like managing scaling, backups, and database tuning for their specific workload.

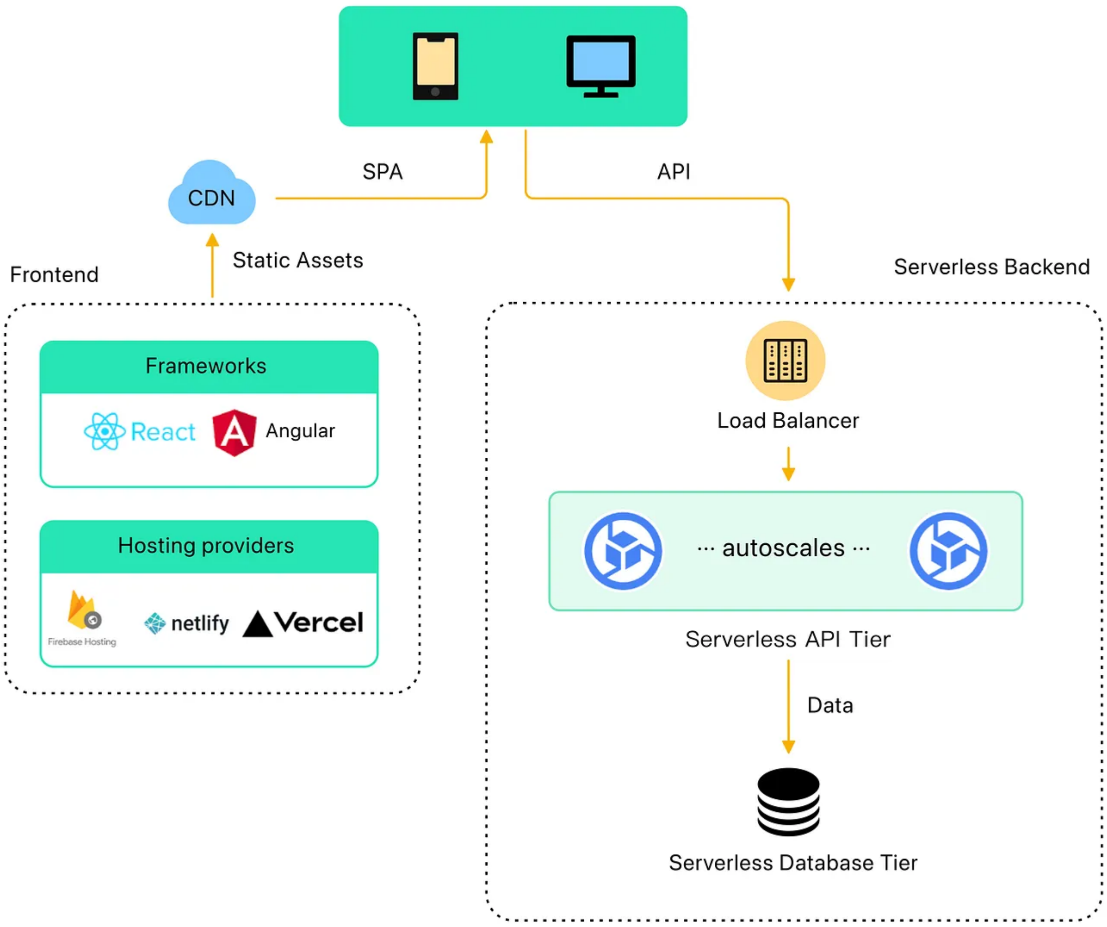
As with many things in software engineering, the choice of the first database is about tradeoffs. There are certain situations where a NoSQL database is a better choice, but for most startups where product/market fit is uncertain, and maintaining maximum flexibility is invaluable, a relational database is likely a better choice.

Modern Startup Stack

By leveraging a production frontend hosting platform, a serverless API backend, and a serverless relational database, developers these days start with a stack that will take them far without investing a whole lot of effort upfront.

With this stack, they are not managing any infrastructure or servers. These offerings have high availability and multi-zone redundancy built in. The providers own the responsibility of scaling the services up as traffic grows.

This modern stack is powerful. It could handle many tens of thousands of users, with some reaching hundreds of thousands. It is a far cry from what we were able to achieve with a single server.



In part 4, we will explore ways to scale this modern stack as traffic grows.

168 Likes · 2 Restacks

16 Comments

Write a comment...

Deepak Mar 4

Thanks for summarizing the pieces of modern software design in clear visual diagrams. Please share the tool you are using for these beautiful diagrams!

Couple of questions:

1. Can you explain how relational SQL provides greater schema flexibility? When you are a startup, more often you will be changing the schema to include new fields as you

iterate on the requirements and was assuming NoSQL would be easier since the row/column structure is not strongly coupled.

2. Can you elaborate more on the proxies and reverse proxies used in modern applications?

LIKE (4) REPLY SHARE

5 replies

Julia Tong Mar 10

Loving it!

I found the API diagram pretty interesting , and worth discovering.

Could you please have a write up to drill down the API gate part?

LIKE (1) REPLY SHARE

14 more comments...