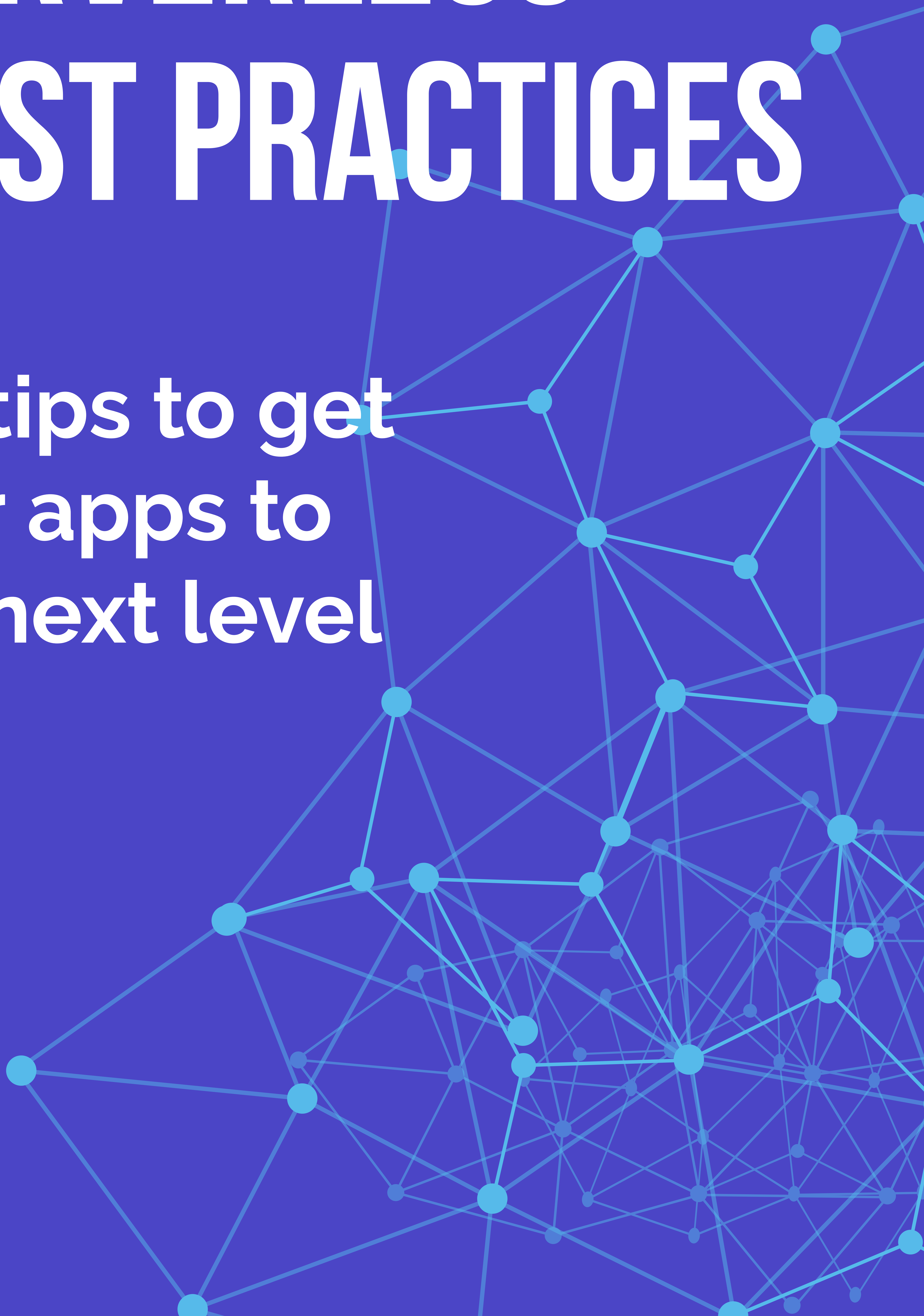
Dashbird.io ++++++++

SERVERLESS BEST PRACTICES

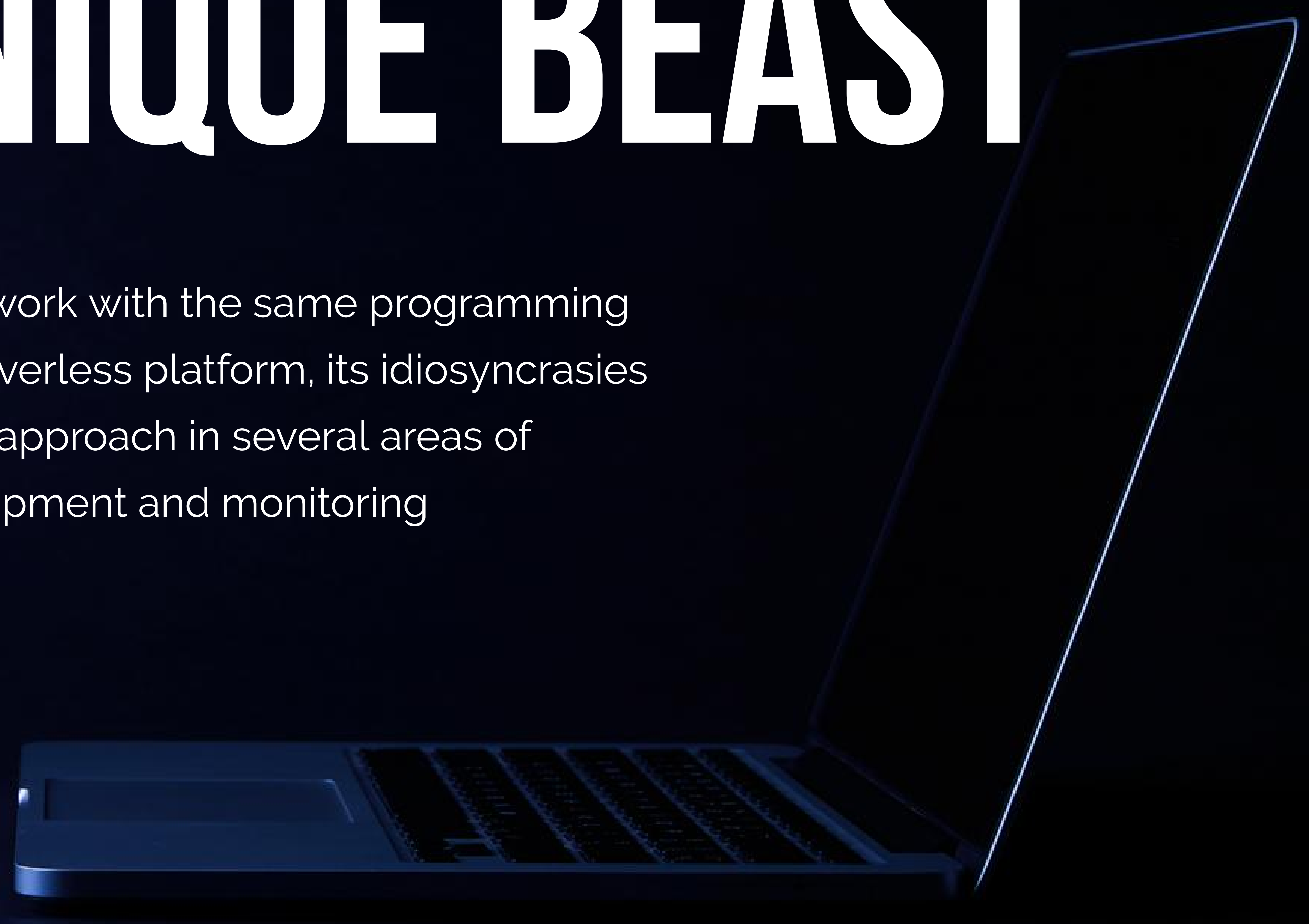
Pro tips to get your apps to the next level



Intro ++++++++

SERVERLESS IS A UNIQUE BEAST

Although we can work with the same programming languages in a serverless platform, its idiosyncrasies require a different approach in several areas of application development and monitoring



Serverless is a different beast in comparison to traditional infrastructures where developers retain the burden of configuring and monitoring their own fleet of machines. Serverless abstracts away all this and while doing so it substantially changes how we should think about architecting, deploying and monitoring applications. The goal of this ebook is to introduce some of the best practices and cover what is needed to appropriately adapt our approach to the serverless paradigm.

The ebook in structured in four sections:

- **Frameworks**: makes a developer life a lot easier to deploy and configure serverless apps
- **Logging & Debugging**: how to best track and fix issues in serverless environments
- **Monitoring**: stay on top of your application performance and behavior to ensure its quality
- **Architectural Patterns**: leverage serverless to produce maintainable, extensible and resilient code

Bear with us on the next pages to take off for a whole new level of application development!

Frameworks

— "I would go so far as to say that there is no minimum level of complexity under which you shouldn't use a [serverless] deployment framework." Paul Swail

Some tasks required to configure and deploy a serverless application would be too time consuming (and actually boring) to do without the help of a serverless framework. A developer gets no benefit from taking care of these tasks manually, so it makes total sense to take advantage of a framework.

Here is a list of tasks that a serverless framework will help you on:
- Bundling and packaging code for deployment
- The actual deployment process (which may take multiple steps in itself)
- Setting environment variables
- Managing secrets
- Configuring endpoints to expose your serverless service as a public API
- Taking care of properly setting the permissions needed by the function
- And more...

The ecosystem for serverless frameworks is quite rich and diverse. Some are focused on specific programming languages and/or cloud providers. Others are runtime and cloud-agnostic. Next is a list of some of the main frameworks, as well as their strengths and weaknesses.

Serverless Framework

Website: https://serverless.com
Repository: https://github.com/serverless/serverless
License: MIT

This is perhaps the most popular framework for serverless environments currently. It's cloud-agnostic and extensible with community plugins. Unless your application requires specific features for special needs not supported by it, we would recommend this framework as the best choice.

Chalice

Website: https://chalice.readthedocs.io/en/latest/
Repository: https://github.com/aws/chalice
License: Apache 2.0

Created and open-sourced by the AWS Lambda team, Chalice is one of the most popular frameworks targeted at AWS Lambda and Python developers. The way it handles declarations of HTTP endpoint handlers resembles a lot how the popular framework Flask works, which makes it a bit easier to migrate Flask applications to the AWS serverless platform.

AWS SAM

Website: https://aws.amazon.com/serverless/sam/
Repository: https://github.com/awslabs/serverless-application-model
License: Apache 2.0

SAM stands for Serverless Application Model. It is a framework developed by AWS that prescribes ways to express a serverless application targeted at AWS Lambda. It then uses AWS CloudFormation to deploy and manage SAM applications in the cloud. Also offering a local command line interface (SAM CLI), it provides a way to simulate AWS Lambda locally, greatly simplifying the development and testing of serverless projects.

Other Contenders

As we said above, the ecosystem is quite rich and we could go over this list endlessly. The purpose of this section was to introduce you to the main serverless frameworks and give a head start for your own investigation. We encourage you to explore options and verify which framework would better suit your needs.

Additional to the list above, we suggest going through this great compilation of frameworks:
https://github.com/anaibol/awesome-serverless#frameworks

Logging & Debugging

— "Anything that can possibly go wrong, will go wrong".
Murphy Law

In serverless environments such as AWS Lambda, developers can't have control over the platform that is running code. That means developers can't rely on metrics being generated by background processes or daemons anymore. Code could be instrumented to send metrics to third-party services in real time, but that means latency will be added to the overall execution. Amazon conducted a study in the past and has found that every 100 milliseconds added to a customer facing request leads to 1% loss in sales.

To properly conduct logging and debugging of serverless applications we must rethink the way we approach these activities. In this section we cover good practices to make sure our apps are on track.

Local Environment

One of the challenges in serverless, especially when using a cloud service provider, such as AWS Lambda, is replicating the production environment in a local machine, as closely as possible.

There are many solutions to tackle this challenge. AWS, for example, offers two ways to replicate Lambda in a local machine:

IDE Plugins
AWS has plugins for both Eclipse and Microsoft Visual Studio IDEs. The Eclipse plugin is more limited, not helpful for local debugging for example, while the Visual Studio one is more comprehensive, helping with development and debugging, but unfortunately it is limited to .NET applications only.

SAM Local
As discussed previously on this e-book, SAM is one of AWS serverless frameworks. It provides a CLI interface to emulate Lambda locally, making it easier to test and debug serverless applications

Each cloud provider will offer its own solutions to help in local testing and debugging. Open source serverless projects, such as OpenFaaS, will also have their own models and solutions for local environment simulation. Depending on the solution chosen for a project, it will determine which are the best options for local environment configuration.

Cloud Logging Services

AWS CloudWatch
CloudWatch is the default AWS solution for logging. It is designed to serve the entire AWS catalog of cloud services. It does its job well, but since it was not designed especially for serverless, it misses many key features desired for this type of platform.

A few of these missing serverless features in AWS CloudWatch:

- CloudWatch keeps multiple invocations logs in a single log stream, which makes it cluttered and harder to debug issues;
- It is not possible to filter invocations by status or type of result; for instance, when investigating an error, it would be handy to filter only invocations that resulted in errors, obviously, but it's not possible in CloudWatch;
- Does not do a great job in identifying cold starts or retries; Lambda has a retry behavior that may cause serious issues, if not considered appropriately; being able to track retry invocations, linked to the original source, is paramount, but CloudWatch misses this need;
- It is very common to have multiple functions working together (as micro services) to accomplish a larger task; being able to debug those functions as a single group is very handy when it comes to narrowing down sources of issues for example, but CloudWatch does not help here;

Azure Log Analytics
Azure logging services goes a bit ahead of CloudWatch, offering not only storage and browsing of logs, but also some analytical services on top of those. It applies machine learning, for example, to monitor application performance and identify opportunities for optimization.

It suffers from the same limitations as CloudWatch in the sense that it's not designed for specifically for a serverless architecture, thus lacking support for the way this type of application needs to be debugged and monitored.

Google Stackdriver
Very similarly to AWS and Azure, Google covers most logging and monitoring needs with the Stackdriver offerings: from logging to monitoring and tracing of cloud applications.

Just as AWS and Azure, as well, Google catalog is not tailored for serverless applications.

Full featured logging services

Since most cloud providers do not offer everything that development teams need for logging and debugging serverless applications, there are a few third-party services available with more advanced offerings. They can save a lot of time and even avoid all sorts of damage (branding, reputation, legal, etc) due to poorly monitored and maintained applications.

Dashbird
https://dashbird.io
Dashbird was designed from the ground up to meet the needs for serverless applications in terms of logging and monitoring. It allows developers to view invocations individually, links all invocations failed and retried, tracks application exceptions and runtime errors, such as timeouts, memory exhaustion, etc.

Dashbird approach doesn't add a millisecond to the function execution time. By collecting logs from CloudWatch and parsing them in a smart way for alerting and reporting purposes, Dashbird empowers developers to monitor applications with ease without compromising performance.

Dashbird also offers some handy features, such as grouping multiple functions into a single group, called "Project", which allows to monitor and debug them as a whole, saving time and gaining visibility over the application stack. By setting custom policies following performance thresholds the application should meet, Dashbird proactively alerts the development team when functions starts behaving in an undesired or unpredicted way.

Another advantage is that pricing is also tailored for the serverless architecture. The service charges proportionally to the amount of data generated in logs, so developers have full control over how much they want to use and pay.

Datadog
https://datadog.com
Datadog is an old player in the logging/monitoring services arena. Although it has traditionally targeted a broader usage of cloud services with general-purpose offerings, only very recently it has started to catch up with the needs of serverless developers.

NewRelic
https://newrelic.com
Similarly to Datadog, NewRelic has been a traditional player in this field, with general purpose offerings for logging and monitoring. It has also made efforts to adapt itself for the needs of teams using serverless, but still lack the tailoring that we find in services purpose-built for serverless applications.

Security

Some types of information are critical to log in serverless applications, so that they are available when it comes the time to act on security breaches. Having critical logs will help, for example, understand which security flaws attackers explored and how to fix them, or build a blacklist of IP addresses, or identify compromised customer accounts.

Below are some examples of information we could classify as critical for logging in a serverless app.

Invocation/Event Inputs
Having the event input data will be important when analyzing a possible security breach. We could retrace the attackers steps and understand which flaws were explored.

Response Payload
Similarly to Invocation Inputs, logging response payloads could also be helpful to analyze and mitigate security breaches. First of all, in the worst case scenario of not being able to stop an attack, we will at least want to know what information is now in possession of the attackers. These logs will answer just that.

Authentication Requests
For applications that have some sort of login/authorization protected area, it's important to log the authentication requests. If the app starts getting too many authorization failures, we would want to have policies (more on the next section) to alert us, since that could be a signal of an attacker looking to crack a user's account.

Monitoring

— "Monitoring Serverless is a new beast in itself. Traditional methods will not work. A new mindset is in order". Adnan Rahic

Tracing

Serverless functions will almost always interact with external services, especially because they intrinsically can't persist data. Whether it's a database, an object storage, a datalake or else, functions need external storage to accomplish tasks that rely on statefulness of data. Other services interacting with functions may be message queues, data streaming processing, authentication mechanisms, machine learning models, etc.

While tracking the entire lifecycle of a serverless function runtime execution is essential for performance improvement, security monitoring, debugging, etc, all these external interactions pose difficulties for that. For this reason, there are solutions especially tailored for instrumenting serverless functions.

AWS, for example, offers a service called X-Ray. It's role is to basically track everything your function runs, not only internally, but also all the interactions with external services in the same cloud or externally (a third-party HTTP endpoint, for instance). As your function runs, it collects valuable insights, such as latency, information exchanged, etc., to help developers fully understand what is happening under the hood, which steps occurred first, what caused a given error, which parts are compromising speed of execution, among other evaluations.

Some monitoring/debugging services, such as Dashbird (discussed above), also integrate with X-Ray (or similar services), so that developers can have everything in the same place: logs, error reporting, alert policies and the entire instrumentation of their serverless apps.

One very important thing to consider when choosing a tracing system for instrumentation is its scalability and availability. Since serverless platforms, by definition, can scale very quickly and offer high availability, the tracing system must be able to cope with the elastic demand of serverless functions.

Performance

Although serverless functions offer virtually infinite elasticity in terms of scalability, that does not mean they should run unmonitored. It is paramount to set up thresholds of performance expectations so that it is possible to determine when something requires attention.

Some of the key measures to look for are:
- Invocations count
- Count of runtime crashes, application failures, cold starts, retries
- Memory utilization
- Duration of executions

Invocations count is obviously essential because it is going to tell developers how many times a function is being used. That's the main factor driving resource utilization and thus costs. The number of failures will the functions health. If a function starts getting too many timeout errors, for example, it's definitely worth developers' attention. The function might rely on third-party HTTP APIs that started to slow down, or maybe the database used by the function is under too much load and needs care. That's also valid for excessive duration times.

Cold starts happen when our function does not have enough containers to serve the number of requests coming in on a given point in time. This forces the underlying serverless platform to spin up a new container – which may take from a few hundred milliseconds to several seconds – while the requester is waiting for a response. There are many scenarios where this is undesirable. If that's the case for our application, we need to detect and monitor cold starts in our stack. Cloud services usually won't provide this information directly, but monitoring services such as Dashbird will.

Memory utilization is an important measure to identify opportunities for optimization. If a given function consistently used only a fraction of the allocated memory, there may be space for reducing resources allocated and saving money on the function execution.

Financial Costs

As stated before, it's important to look out for opportunities to improve resource utilization and reduce our serverless stack bill.

Usually cloud services like AWS won't provide all the data we need for the analysis, but there are services like Dashbird where we can look, for example, how much memory has been used by a function, over a given period of time. It gives us the average, minimum and maximum utilization, making it easier to identify optimization opportunities.

These services will also provide cost statistics broken down by function, instead of the entire serverless stack – which is usually how cloud services will provide in their billing statements. Being able to narrow down the analysis on a per-function basis is essential.

Policies

Setting custom policies based on the expected behavior of our functions is also of high importance. If we expect, for example, that a function should take around 5 seconds to finish requests, we would want to know when it starts taking 20, or maybe 50 seconds, for whatever reason. By having custom policies in an advanced monitoring service, we can have it alerting ourselves so that our development team can be proactive and immediately jump over the potential issue.

Architectural Patterns

— "Our understanding of how and when to use Serverless architectures is still in its infancy. We're starting to see patterns of recommended practice occur, and this knowledge will only grow."

Martin Fowler

API Gateway

An API Gateway service - either managed by a third-party provider or custom deployed - will provide a single entry point for external requests, so that clients talk to the gateway, not each service individually. Additional benefits would be having a layer of security against malicious attacks embedded in the infrastructure behind the gateway, some concurrency handling to limit individual clients ability to consume services, as well as authentication/authorization control logic.

Although an API Gateway is applicable to any cloud platform, serverless may benefit particularly from this pattern, especially when adopted together with a microservices approach.

AWS, for example, offers an API gateway service that integrates with Lambda, their serverless platform. Every major cloud service will have an offering to cover this area. There are also third-party services you can use.

Microservices vs. Monolithic

One advantage of a serverless platform is that we can create multiple functions that work together composing a single major system. Instead of having one function taking care of everything to perform a task (monolithic approach), it is usually a good idea to break down the service into microservices, so that each of those minor functions have a small job to do, and they do them well.

A microservices approach also contributes to better code maintainability, reusability and portability, which are great indicators for the overall project health.

Idempotent Functions

Depending on the flow of our system, retries can be harmful. For instance, let's imagine a function that is responsible for adding a user to the database and sending a welcome email. If the function fails after creating the user and gets retried you will have duplicated entries in the database.

A good way to overcome this is to design our functions to be idempotent. Idempotent functions are functions with a single task, which either succeeds or can be retried without any damage to the system. You could redesign the aforementioned function using AWS Step-Functions. First being the function responsible for adding the user to the database and as a second step, another function sends the email.

Event-based

An event-based approach relies on being able to have our serverless functions automatically triggered upon a given event occurrence. For example, AWS Lambda integrates with S3 (object storage): we can configure it to automatically invoke a certain function when a new object is stored in a predetermined bucket. AWS provides other integrations with Lambda that can be very helpful, such as DynamoDB (NoSQL key-value database), Kinesis (stream processing) and more.

Other cloud services will have similar integrations, but from our observations, AWS is the cloud service showing the highest commitment to advancing their serverless platform, so we should expect a better offering from AWS in terms of integration as well.

This pattern can be useful and increase productivity in the application development since we can build on top of interfaces already developed and matured by the cloud service provider and quickly come up with workflows that can be complex, but yet simple and reliable.

Message-based

This architectural pattern resembles a bit the Observer pattern in object-oriented programming. In essence, when we need to coordinate multiple serverless functions, it's usually better to have them communicate with each other through messaging systems than directly invoking each other.

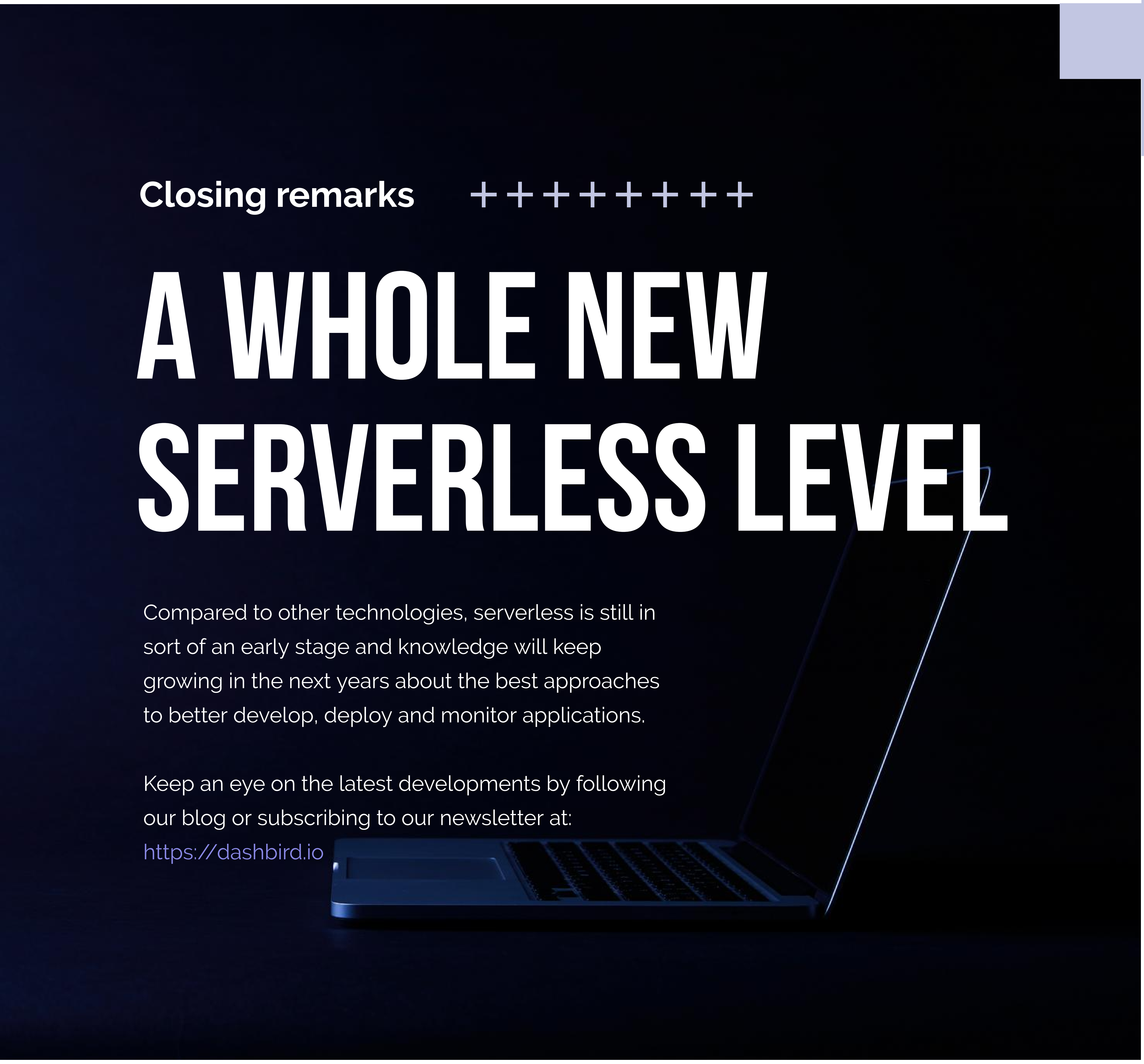
In a message-based architecture, we create message queues for each communication flow needed by the application. One or more functions will publish to the queue, and other function(s) will subscribe to the queue and consume those messages. When there's a need to change the workflow, it's just a matter of changing the queue subscriptions, which usually simplifies the adjustment and reduces risks of unexpected issues.

Fan-in / Fan-out

This pattern breaks down a process over multiple tasks that are taken care of by individual workers (fan-out) and later results from each worker can be grouped into a single consolidated result (fan-in).

Fan-in/fan-out is especially suitable to a serverless environment because each function invocation runs in an isolated container with its own resources, which opens up opportunities for processing improvements.

CPU-intensive tasks are usually good candidates for this approach. IO-bound tasks can also benefit to a fan-in/fan-out strategy, but not as straightforward. With IO-bound tasks, it might be cheaper to use multithreading to optimize resource utilization. It will greatly depend on the runtime: programming languages like C and Go are good for this type of task and could handle concurrency very well in a single serverless function. Other languages, such as Python, which are not as optimized for concurrency and parallelism, may end up struggling. In this case, fan-out of IO-bound tasks to multiple serverless functions could be a good option.



Closing remarks ++++++++

A WHOLE NEW SERVERLESS LEVEL

Compared to other technologies, serverless is still in sort of an early stage and knowledge will keep growing in the next years about the best approaches to better develop, deploy and monitor applications.

Keep an eye on the latest developments by following our blog or subscribing to our newsletter at: https://dashbird.io

We trust this ebook shed some light on issues and topics you should be aware and get deeper in order to appropriately approach the software development cycle for a serverless platform.

Dashbird was especially designed from the ground up to provide the best monitoring and debugging experience for serverless developers using AWS Lambda. If you are looking for a way to professionally monitor your serverless applications in production, you should definitely check it out. You can try our full set of features for a trial period or even use our service entirely free, in case your needs aren't that big.

Dashbird: https://dashbird.io