SERVERLESS COMPUTING

DEVELOPER EMPOWERMENT REACHES NEW HEIGHTS

Digital transformation is at the top of mind across every organization, regardless of how rooted in technology they may be. Continually keeping up with the rapid pace of innovation is essential to maintain competitive advantage, and smart companies have recognized that it's the developers that hold the keys to success with their ability to build and ship software. Technological advancements across the entire cloud ecosystem, complemented by cultural shifts within organizations, have helped remove roadblocks from the software development lifecycle, further paving the way for developers to deliver continuous innovation.

Now imagine a future where the notion of provisioning and managing infrastructure resources is completely removed from the development process – in production and at any level of scale. Distributed systems are constructed to react to surrounding environments by spinning up ephemeral compute resources that process dynamic workloads on demand. For the developer, the only interface is APIs – the systems are intelligent enough to automate the workloads entirely behind the scenes, while being fully aware of the surrounding environments at all times. While this may appear out of reach, it represents a future that has already arrived through a new architectural pattern, aptly named "serverless".

This white paper details the key benefits and implications of a serverless architecture by walking through the entire development lifecycle. While a key tenet of this theme is abstracting away the underlying operations, having a clear understanding of what's happening behind the scenes will allow developers, architects, and business owners to make educated decisions about the future direction of their systems and applications.

TABLE OF CONTENTS

3 Introduction to Serverless Concepts
4 The Evolution of the Modern Cloud
6 Preparing To Go Serverless
10 The Serverless Development Lifecycle
14 Why Serverless Matters

ABOUT THE AUTHOR

IVAN DWYER

Ivan is Head of Business Development at Iron.io, working with partners across the entire cloud technology and developer services ecosystem to form strategic alliances around real world business solutions.

© 2016 Iron.io, Inc.

INTRODUCTION TO SERVERLESS CONCEPTS

Despite the implications, a serverless architecture doesn't mean getting rid of the data center through some form of black magic that powers compute cycles in thin air. At its core, the concept promises a serverless experience for developers; as in never having to think about provisioning or managing infrastructure resources to power workloads at any scale. This is done by decoupling application components as independent microservices that automatically run when a predetermined event occurs. These events arise from a variety of sources – from an application such as a webhook, in the real world such as a sensor capture, within a system such as a database change, or on a schedule such as a cron job. The automated workflow happens entirely behind the scenes without any need for manual intervention – spin up, execute, tear down. Rinse and repeat at massive scale.

It's important to note that nothing about this pattern discounts the importance of proper infrastructure management. The primary goal is to abstract the complexities away from the development process, which in fact places even more emphasis on finely tuned system administration in order to enable such automated capabilities. DevOps best practices have set the stage within the modern enterprise by promoting continuous integration and delivery models that allow for dozens, or even hundreds of deploys per day. What's different about the serverless architecture is that the underlying operations are manipulated via API instead of through scripts. In shifting the configuration from the systems layer to the application layer, the mantra Infrastructure as Code thus becomes Infrastructure in Response to Code.

IN SHIFTING THE CONFIGURATION FROM THE SYSTEMS LAYER TO THE APPLICATION LAYER, THE MANTRA INFRASTRUCTURE AS CODE THUS BECOMES INFRASTRUCTURE IN RESPONSE TO CODE.

KEY BENEFITS

FASTER TIME TO MARKET

Developers with a clear path to delivery are able to focus their core efforts on building the features that drive the business forward in shorter release cycles.

MORE EFFICIENT USE OF RESOURCES

Breaking down applications into smaller, discrete units of compute that only use up resources when needed makes the most of what's available and reduces overall spend.

LOWER TOTAL COST OF OWNERSHIP

Investing in workload optimization and intelligent systems design up-front leads to lower administrative costs over time, and sets the foundation for future growth.

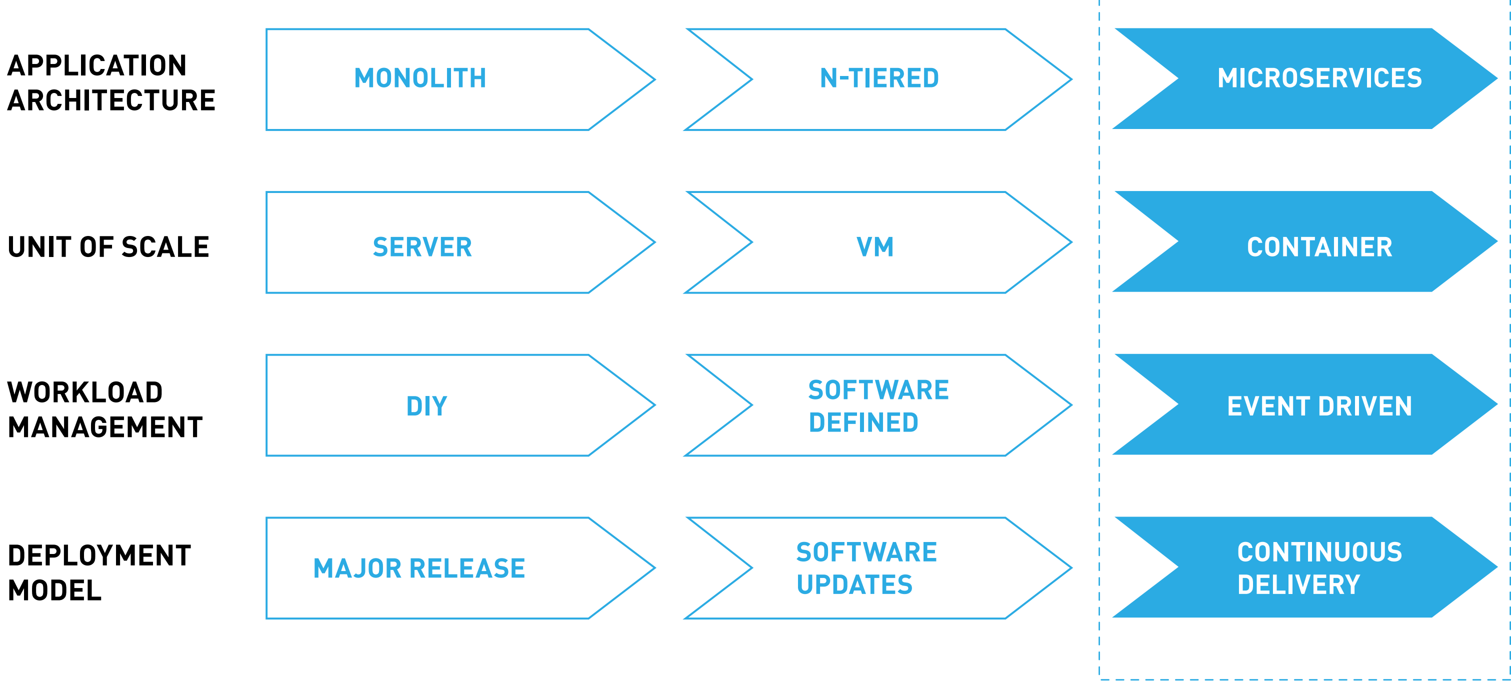
LESS MAINTENANCE

Fully automated workflows removes the need for manual oversight, enabling a fully streamlined development lifecycle.

© 2016 Iron.io, Inc.

THE EVOLUTION OF THE MODERN CLOUD

While many of the associated concepts and patterns have existed for some time, it's been the convergence of a few parallel evolutions that have truly brought forth this serverless future. Without any of the following, we wouldn't be able to have this conversation.



© 2016 Iron.io, Inc.

MICROSERVICES

Born from the challenges of running distributed applications at scale, microservices has emerged as a dominant pattern for breaking apart applications into more manageable pieces. By decoupling application components as independent services that each perform a single responsibility, individual workloads become portable enough to distribute via automated workflows.

CONTAINERS

Portability is meaningless without consistency across environments. Container technologies such as Docker have taken the industry by storm through effective process isolation. Containers allows microservices to be developed, deployed and executed anywhere with the confidence that the runtime will be as expected.

EVENT-DRIVEN

The rise of all kinds of connected devices has led to a transition from the traditional request/response model of instantiation towards more responsive models. With the ability to react to device, system and application changes automatically, containerized microservices run entirely behind the scenes without manual intervention.

DEVOPS

These reactive workflows then need to be complemented with automated provisioning and deprovisioning to fully realize a serverless experience. The goal is to abstract away everything but the API, but that means the underlying system must be capable of handling all the infrastructure operations beyond just scale, including health monitoring and failure handling.

Serverless computing represents the intersection of these modern cloud movements; where architectural patterns meets development culture in a way that delivers true value to the business. Smaller development teams who can confidently claim ownership and responsibility enable the continuous innovation required to stay competitive.

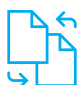
> SMALLER DEVELOPMENT TEAMS WHO CAN CONFIDENTLY CLAIM OWNERSHIP AND RESPONSIBILITY ENABLE THE CONTINUOUS INNOVATION REQUIRED TO STAY COMPETITIVE.

© 2016 Iron.io, Inc.

PREPARING TO GO SERVERLESS

Any new development trend capturing headlines needs to be backed up with real world solutions in order to escape from the hype cycle to the enterprise. Given that serverless computing is the culmination of several parallel evolutions that have already taken root in today's enterprises, it is poised to become the next significant technology shift. With that in mind, let's examine what it takes to implement these patterns into our own applications.

IDENTIFYING THE RIGHT CANDIDATES



BACKEND TRANSACTIONS

These include individual user or device actions such as processing a credit card or sending an email. When applications demand real-time responses, any processing should be offloaded to the background, so it doesn't interfere with the user experience. Individual transactions are easy to decouple as containerized microservices, at which point they can be triggered to execute asynchronously without the need for an immediate response.

BATCH PROCESSES

Likely candidates for serverless batch processing are heavy lifting workloads for crunching a large data set or encoding a multimedia file. The most common pattern in the history of computing still has a strong

© 2016 Iron.io, Inc.

place in the modern cloud. What's changed the most aside from the compute environment is the need for a manual start. An event can trigger a workflow that runs batch jobs concurrently across a pool of available compute resources.

DATA PIPELINES

A data pipeline may include extraction, transformation, and loading of data such as analyzing machine data for an IoT system or delivering filtered logs to a data warehouse. The explosion of connected devices has brought forth exponential growth in the number of data sources and sheer volume, requiring an awful lot of processing power to extract meaningful insight. Getting data from source to destination happens in the background, with any logic inserted along the way.

BOTS

With automated workflows being a key theme with serverless computing, a natural extension is automated services. Chatbots or integration hooks are meant to run independently in response to events - user or system generated. The input is taken in as a payload, the process is executed, and then the results are delivered back.

SCHEDULED JOBS

Examples of scheduled jobs could be regular occurring jobs such as a daily web scraper or weekly email blast. Cron jobs are a staple of any system, but are also easy to forget and hard to manage. With a serverless architecture, a schedule is simply a form of event trigger, used to kick off any type of background workflow as shown here.

© 2016 Iron.io, Inc.

UNDERSTANDING EVENT TRIGGERS

The event-driven methods for instantiating workloads means giving extra care to what happens once invoked. Real world events have a tendency to pile up, which can quickly lead to system overload or resource maxing if not careful. If you treat each event as an API endpoint, then you'll want to ensure you have the right policies in place for authentication, authorization, rate limiting, error handling, and more. First things first, it's important to understand where different events can be sourced. When triggered, the job is posted to a queue that either pushes the message to a running worker node, or holds the message until a worker node is available.

SYSTEM CHANGES

Infrastructure resources can be wired to capture internal events, which are then passed as messages to trigger a workflow. For example, this can be a notification when a file is uploaded or when a database record is changed.

APPLICATION CHANGES

Applications can incorporate webhook urls as a trigger method. Many services now incorporate webhooks across events, allowing developers to map the url to an API endpoint that automatically invokes a workflow.

STREAM

Data as a stream of events is common in modern applications, with a strong need for heavy processing capabilities at the end. Workers can be configured to concurrently process data across a pool of compute resources.

SCHEDULE

Regular occurring jobs such as a daily web scraper or weekly email blast are a staple of any application, but are also easy to forget and

© 2016 Iron.io, Inc.

hard to manage through cron for example. In this context, a schedule is simply a form of event that triggers the workflow in the background.

MANUAL

Any job can be invoked manually, either via direct API call or CLI. Aside from testing purposes, some jobs are built to be executed on demand such as a database query or system cleaner. When invoked, the job is placed in the queue just as any other event-driven method.

NAVIGATING THE VENDOR LANDSCAPE

While many of the associated concepts have existed for some time, the serverless trend really caught on with the introduction of AWS Lambda in late 2014. Not to be outdone, Microsoft and Google recently announced their own versions, Functions and Google Functions, respectively. Although you may think the clear choice would be to pick the infrastructure provider where your applications are running, there are some additional points to consider.

While a key benefit to an IaaS-native solution is the ability to leverage internal system events to trigger the automated workflows, this could also be seen as a strong deterrent in terms of vendor lock-in at the technology level. Once you start coupling your business logic to the surrounding system, you are stuck in that environment – much like the legacy middleware era we've done such a great job escaping. Unless you are confident that you are going to run in the same, single cloud provider for the next 5-10 years, it's worth considering what it means to lock-in at that level.

For a solution independent of any sole infrastructure provider, Iron.io offers a container-based serverless computing platform that is available in any cloud, public, private, or hybrid. As more and more developers adopt Docker as the preferred development environment, it becomes more natural to work in this style with a portable, container-based solution.

© 2016 Iron.io, Inc.

THE SERVERLESS DEVELOPMENT LIFECYCLE

Shifts in application architecture mean making adjustments to the development process in order to reap the benefits. The serverless experience we're looking to achieve will kick in once the jobs are developed and properly prepared for automated execution. Iron.io provides developers and end-to-end platform for powering job-centric workloads through an easy to use API and interface. The following is a walkthrough of the Iron.io development process, with some best practices along the way.

1. BUILDING THE JOB

Developing with Docker is a breeze as you can work across multiple languages and environments without clutter or conflict. When your job logic in place, you specify the runtime by writing a Dockerfile that sets the executable, dependencies, and any additional configuration needed for the process.

TIPS

Choose a lightweight base layer. This can be a minimal Linux distribution such as Alpine or Busybox. Read about microcontainers on the Iron.io blog.

Keep the layers to a minimum. Don't run an update on the OS and consolidate RUN operations inline using && when possible.

Limit the external dependencies to only what's needed for the process itself, and vendor ahead of time so there's no additional importing when the job is started.

2. UPLOADING THE IMAGE

Each serverless job is built as a Docker image and pushed to a registry, where it can be pulled on demand. This can be a third party public image repository such as Docker Hub, Quay.io, or your own private registry.

TIPS

Incorporate the job code into a CI/CD pipeline, building the container image and uploading to a repository.

Version your images using Docker tags and document properly. Don't rely on :latest as what should always run.

© 2016 Iron.io, Inc.

3. DETERMINING EVENT TRIGGERS

Once the job image is uploaded, it becomes ready to execute, either through an event or manually. With Iron.io, events can be set via API, native SDK, or dashboard. The automated workflow will then occur when the event happens or when the job is explicitly called.

TIPS

Map each job to your API, at a minimum within your documentation, but you can also set endpoints for direct requests. Using an API Gateway is a common way to manage events and endpoints across systems.

Use a load balancer for synchronous requests and a message queue for asynchronous requests to throttle and buffer requests when load is high. With Iron.io, a message queue is built in to the system.

4. CONFIGURING THE RUNTIME ENVIRONMENT

Operational complexities such as service registration and container orchestration are abstracted away from the development lifecycle; however, it is recommended to not just “set it and forget it” when running in a production environment.

TIPS

Profile your workloads for their most optimal compute environment. For example, some workloads are more memory intensive and need more resources allocated to the container.

Set how many concurrent jobs can execute at any given time. This can help keep costs down and ensure you don’t overload the system.

Determine what happens when the job fails. If you want to auto-retry, set the maximum number of times with a delay in between.

5. SECURING AND MONITORING THE JOB

To be production-grade, wrapping the environment with proper security and monitoring is essential. Given the levels of abstraction that serverless computing provides, it’s even more important to gain insight into what’s happening behind the scenes.

TIPS

Payload data should be encrypted at the source and then decrypted within the job code itself. Public key is a common technique in this scenario.

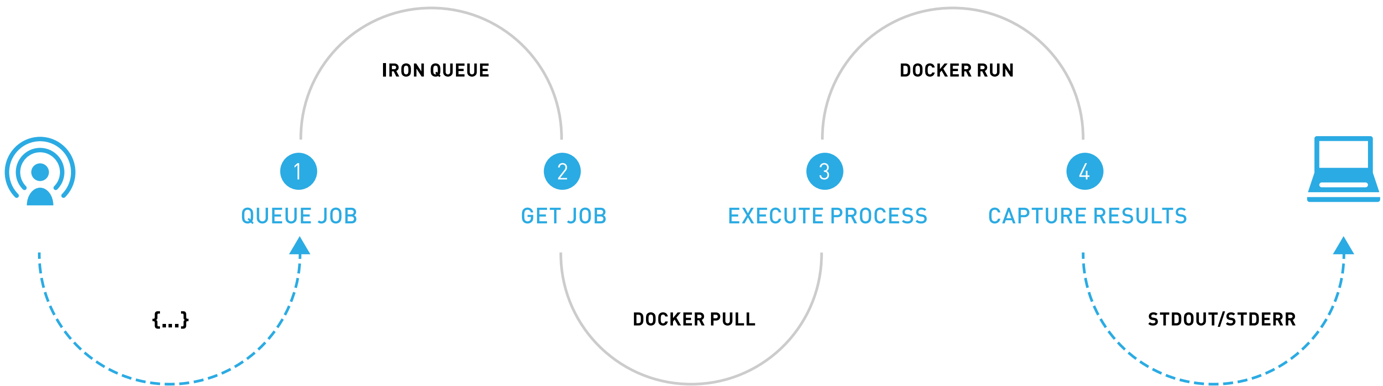
Connections to databases from within a job process that are outside the network should be secure, either through a VPN or IP whitelisting.

Inspect stdout and stderr for each job process. You can pipe these logs to syslog or a 3rd party logging service.

© 2016 Iron.io, Inc.

UNDERLYING OPERATIONS

Even though the end-to-end operations are meant to be abstracted away from the developers, it's worth knowing what's happening behind the scenes to have a clear understanding of the lifecycle of your workloads. When an event triggers an Iron.io job, the following process takes place:



1. QUEUE JOB
The job is posted to a message queue, which acts as the temporary store, holding the job in a persisted state until completed. The message includes the job metadata, runtime parameters, and any payload data to be passed into the job process itself.

2. GET JOB
Worker nodes are continually monitoring associated queues for new jobs. When capacity is available, the worker pops a job off the queue and pulls the associated Docker image from its registry if a cached version of the image isn't already on the machine.

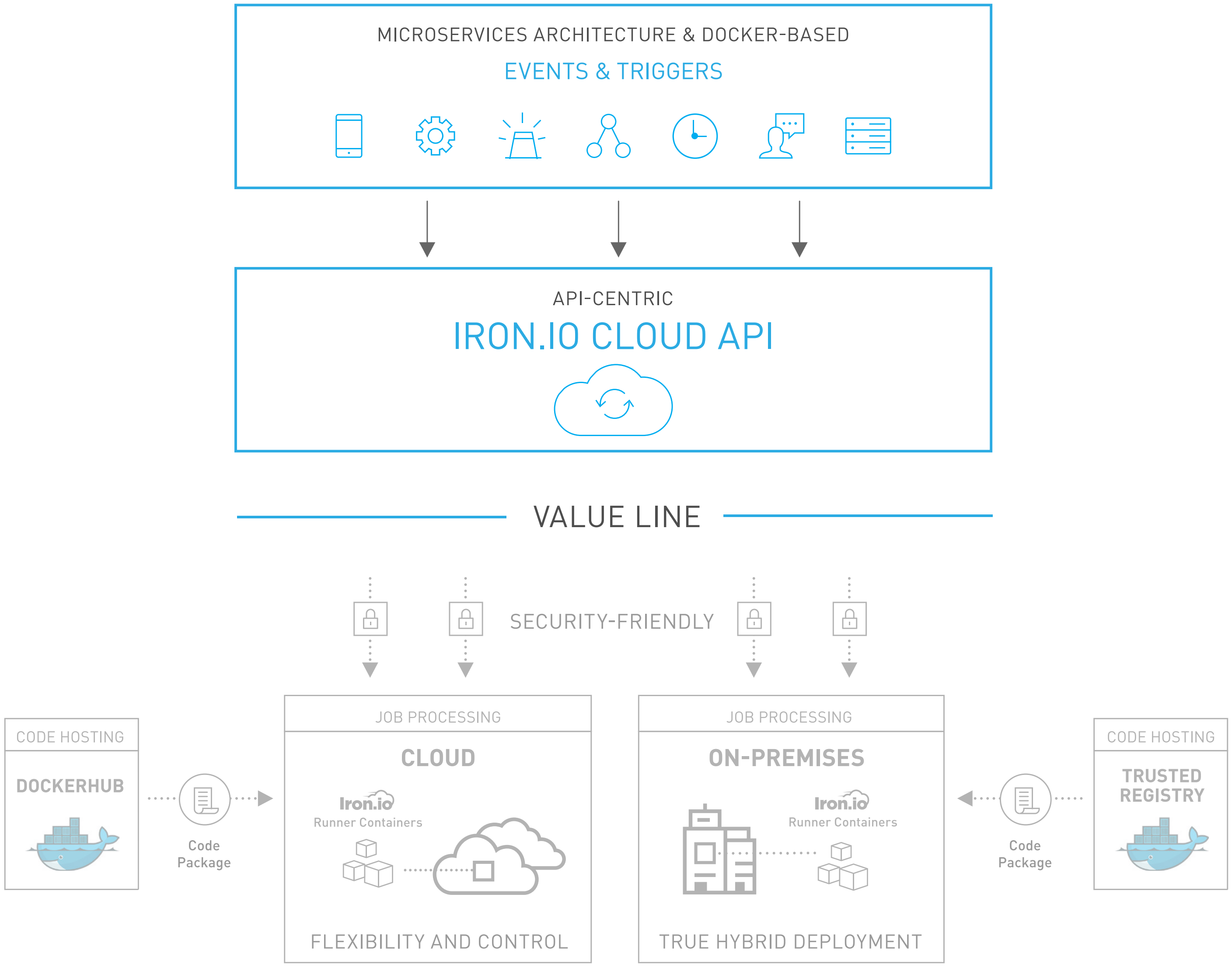
3. EXECUTE PROCESS
The worker then runs the container process with its given resource allocation. When the process is complete, the container is gracefully stopped and the resources are freed up for another job.

4. CAPTURE RESULTS:
The container's stdout/stderr logs are captured and delivered back to the system or a 3rd party service for further inspection. If the process fails or timeouts, the job is placed back in the queue where it can be retried or canceled.

© 2016 Iron.io, Inc.

THE IRON.IO PLATFORM

Iron.io operates an enterprise-grade Platform as a Service specifically meant for powering job-centric workloads throughout their entire lifecycle. Working with Iron.io is a serverless experience because the only interface to developers is an API. If you follow the software development lifecycle and best practices put forth in this whitepaper, you will be able to extract the true value from the underlying technology, which results in true meaning for the business.



© 2016 Iron.io, Inc.
Iron.io

WHY THIS MATTERS

The explosion of container technologies and microservices-oriented development patterns have shaken up the entire IT ecosystem, forcing enterprises and vendors alike to modernize how applications are built and delivered. Much of the initial rush has been focused on the infrastructure layer, such as orchestration of containers across a distributed environment effectively. While an incredibly important topic worthy of its own studies, orchestration only provides the foundation for the end goal of developer empowerment. What this new serverless trend has done, and done very well, is focus on the developer experience at the application layer.

This intersection between the infrastructure layer and the application layer is where Iron.io sits, further closing the gap between APIs and containers by improving the developer experience while removing the operational complexities. The value to any organization is the ability to keep the focus where it matters most - continuous innovation.

> THIS INTERSECTION BETWEEN THE INFRASTRUCTURE LAYER AND THE APPLICATION LAYER IS WHERE IRON.IO SITS, FURTHER CLOSING THE GAP BETWEEN APIS AND CONTAINERS BY IMPROVING THE DEVELOPER EXPERIENCE WHILE REMOVING THE OPERATIONAL COMPLEXITIES.

© 2016 Iron.io, Inc.

© 2016 Iron.io, Inc.