ServerlessRPS
A Continuous-Deployment, Serverless Rock-Paper-Scissors-Over-SMS Reference Implementation Using Amazon Web Services

Tyler Ross
Gianforte School of Computing, Montana State University
357 Barnard Hall, P.O. Box 173880
Bozeman, Montana 59717-3880, United States
tyler.ross2@student.montana.edu
TWR@TWR.name

REVISION 1.1 (April 30, 2021)

So-called "serverless" platforms, commonly realized as Functions-as-a-Service (FaaS), are an increasingly popular way to implement highly scalable applications without the effort or cost associated with conventional platforms. FaaS and, indeed, cloud platforms themselves – Amazon Web Service (AWS), Google Cloud Platform (GCP), etc. – represent such a departure from traditional deployment models that both application architectures, and development workflows require significant changes. We present a complete, serverless application reference implementation, including a GitHub-integrated continuous deployment pipeline, on the AWS platform – using AWS CloudFormation for resource provisioning and management. Once deployed, the application and toolchain described herein provide an extensible foundation for serverless development, and the toolchain may be reused with other SAM-based applications. Particular effort has been taken to maintain the development convenience of SAM CLI deployments, while introducing a production-ready deployment pipeline.

1. Introduction

We begin by motivating our work in the following section. We then describe the architecture of our solution – both "toolchain" and "application" – in Section 2, and conclude in Section 3.

Discussion of application implementation details (game logic, data structures, etc.) is deferred to Markdown documentation alongside the source code: `https://github.com/TylerWRoss/ServerlessRPS`. A walk-through of the deployment and decommissioning processes, as well as the work needed to apply the Toolchain Stack to an arbitrary AWS SAM application, is also provided. Please refer to `README.md`, in the GitHub repository, for an overview of the documentation. Should the repository be unavailable, please contact the author.

1.1. *Motivation*

For the sake of brevity, we will not motivate serverless platforms themselves. Instead, suffice it to say serverless is an area of active interest, and persistent contention, in

both academia and industry. We intend our work as a basis for others to investigate and begin learning the technology, and defer discussion of serverless in general to existing, excellent literature: Castro et al. [4], Shafiei et al. [6], and Eismann et al. [5].

For this work, we focused on Amazon Web Services, and the offerings of other providers were not considered. Though many tutorials and examples, both AWS-official, and third-party, address the development of serverless applications, there is a neglected middle-ground between the trivial 'Hello World' examples, and the opaque, ill-explained 'Serverless for the Enterprise' solutions.

Solutions of the former flavor generally take the form of simple AWS SAM examples. These are sufficient to develop and deploy a serverless application, and are therefore excellent learning resources. However, they neglect the deployment workflows, and security measures, desirable in production systems. For example, such applications typically provide no mechanisms with which to prevent privilege escalations in the event of a compromise. Such solutions are therefore useful for learning serverless application development, but insufficient for learning best practices, and preparing for the industry.

On the other hand, solutions which do address industry and production needs, such as AWS and Trek10's jointly-written and -published "Serverless CI/CD for the Enterprise on AWS" (Warzon et al. [7]), introduce excess complexity, with insufficient explanation. The aforementioned CI/CD example involves multiple pipelines, AWS accounts, and separate, automatically-managed test and production deployments. Furthermore, though a thorough deployment guide is provided, the architecture of the system is described only in passing. Though the solution closely mirrors the type of workflow, and separation of concerns, an actual enterprise might employ, the extreme complexity makes it an impractical introduction to serverless CI/CD systems.

To address this gap, we propose an extensible, separable solution consisting of a "toolchain stack" and an "application stack" – each deployed via AWS CloudFormation. To facilitate rapid development, the "application stack" may be deployed, or updated, separately of the toolchain, using the Serverless Application Model CLI (SAM CLI). Furthermore, as implemented, SAM CLI and "toolchain" deployments may be used interchangeably on the same "application stack." The goal of this architecture is to introduce the workflows and best practices demanded by industry and production environments without compromising the convenient, approachable development workflows of existing AWS SAM examples. In fact, the "application stack" (an implementation of Rock-Paper-Scissors) is independent of the toolchain, and may be substituted for any SAM-based application – with the caveat that IAM roles/policies in the toolchain must be updated to reflect the resources of the application. A key contribution of this work is the effort to describe the components of both stacks, and the nuances (i.e., IAM permissions boundaries) relevant to a secure production environment.

2. Architecture

Shown below is a high-level diagram of ServerlessRPS (SRPS), showing the two CloudFormation-deployed "stacks," and the external actors of the system: a developer, who invokes deployments via GitHub or the SAM CLI, and a user, who interacts with the application via SMS messaging. In the following section, we depict and describe the two stacks.

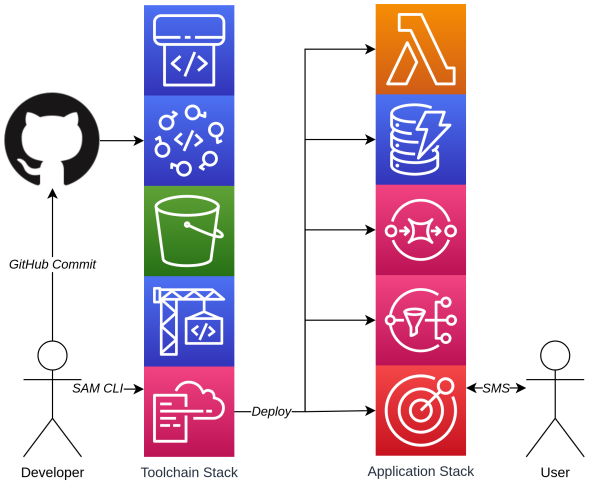

Fig. 1. ServerlessRPS Toolchain and Application Stacks

2.1. *Overview*

The two main high-level components of the system are both AWS CloudFormation stacks: the "toolchain stack" (referred to hereafter as the Toolchain, or Toolchain Stack) and the "application stack" (hereafter, the Application Stack). The Toolchain Stack is responsible for building, deploying, managing, and *constraining* (via IAM "permissions boundaries" – we'll discuss this later) the Application Stack. The Application Stack is responsible for provisioning and configuring the application's resources – lambda functions, databases, queues, etc.. Together, the two stacks manage the complete lifecycle and resources of the application. For development flexibility, though, the Application Stack may be deployed and updated independent of the Toolchain. This separation allows for faster, more flexible development, while still supporting robust, repeatable production deployments.

Only two aspects of this system cannot be automatically configured: the CodeStar Connection to GitHub, and the toll-free number used for SMS messaging between the user and application. The former requires explicit authorization on the

GitHub side of the connection, precluding full automation. The latter must be requested and configured manually, because AWS Pinpoint does not currently support automatic phone number provisioning, or configuration [3]. We discuss the process of deploying SRPS, as well as decommissioning a deployment, in a README which accompanies the source code (linked in Section 1).

2.2. *Toolchain Stack*

The Toolchain Stack sits at the top of this system. Foremost, it defines an AWS CodePipeline Project – AWS' managed continuous delivery service. The CodePipeline Project connects a number of other AWS services into a continuous delivery pipeline. In the Toolchain Stack, the CodePipeline combines a CodeStar Connection (providing GitHub integration), an S3 Bucket, a CodeBuild Project, an IAM Role, and an IAM Policy. Each of these resources, except for the IAM Policy (the aforementioned "permissions boundary," to be discussed later), is shown below in Figure 2.

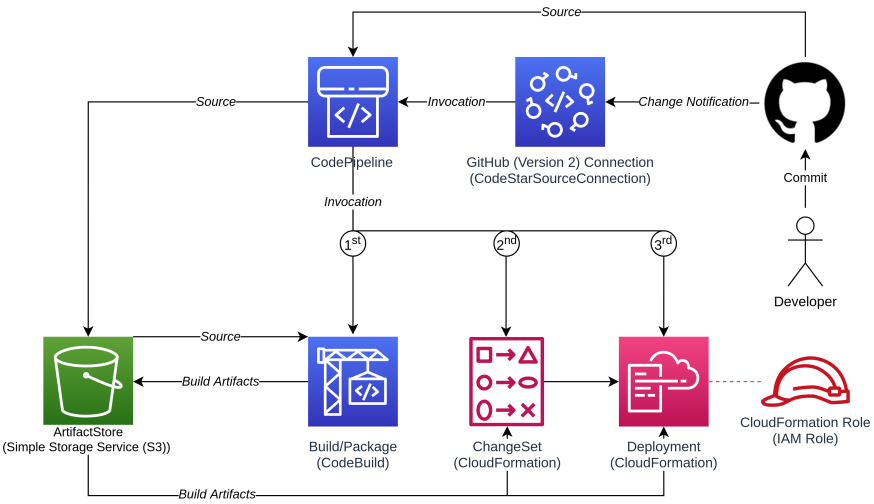

Fig. 2. Toolchain Stack Resources

The CodeStar Connection to GitHub – specifically a "Version 2" connection – provides repository access control (i.e., access to private repositories), and monitors the configured repository for changes. GitHub "Version 1" connections required a separate webhook, configured on the GitHub repository, to receive change/commit notifications. This is no longer necessary, and the integration is now implicit [1].

When a commit occurs to the configured repository, the CodeStar Connection invokes the CodePipeline.

When invoked, CodePipeline reaches out to the GitHub repository, via the CodeStarSourceConnection, retrieves a copy of the application source, and stores the source in the S3 Bucket. CodePipeline then invokes the CodeBuild Project, which retrieves the source, packages it for deployment to AWS Lambda, and stores the build artifacts back into the S3 Bucket. Finally, CodePipeline prepares a CloudFormation "change set," and then deploys it – this CloudFormation deployment creates the Application Stack.

These CloudFormation steps – and therefore the Application Stack – are constrained by an IAM Role, named the "CloudFormationRole." This role defines the resources which may be provisioned and managed by the Application Stack. Ensuring, for example, that a compromised GitHub repository – a commit to which would invoke this CodePipeline – cannot be used to provision unauthorized resources (e.g., expensive EC2 GPU instances for cryptocurrency mining).

Because CloudFormation is given the permissions to create IAM roles and policies (for example, to configure the Execution Role of the application's Lambda Function(s)), an additional safeguard is required to prevent privilege escalation (e.g., attaching the AdministrationAccess Policy to the CloudFormationRole). This additional safeguard is an IAM "permissions boundary." Permissions boundaries are special policies which define upper bounds on the permissions of a role. To prevent privilege escalation, the following statement is used in the CloudFormation Role.

```
- Action:
  - iam:AttachRolePolicy
  - iam:CreateRole
  - iam:DeleteRolePolicy
  - iam:DetachRolePolicy
  - iam:PutRolePermissionsBoundary
  Condition:
    StringEquals:
      iam:PermissionsBoundary:
        Fn::Sub: arn:${AWS::Partition}:iam::${AWS::AccountId}:
        ↪ policy/${AppId}-${AWS::Region}-PermissionsBoundary
```

This statement uses a condition to enforce that created, or modified, roles have the specified permissions boundary. Under this condition, an attempt to, for example, assign the AdministratorAccess policy to a role – and therefore escalate privileges – will confer, at most, the permissions of the permissions boundary policy. As implemented, the PermissionsBoundaryPolicy grants full access to the Application Stack's Lambda Functions, SQS Queues, and DynamoDB Tables, and grants only "SendMessages" permission on the Application's Pinpoint App. These are slightly more permissions than actually needed by the application, but sufficiently restrictive as to ensure a compromised GitHub repository, or compromised application

instance, cannot escalate to an AWS account compromise.

2.3. *Application Stack*

2.3.1. *Deployment*

The Application Stack is defined using the AWS Serverless Application Model (SAM), an extension of AWS CloudFormation designed specifically for building serverless applications. Using an extended CloudFormation template system, application resources and configuration can be deployed by automated systems, like CodePipeline, or via interactive interfaces, such as SAM CLI. In the case of SRPS, SAM CLI may be used to deploy the Application Stack, independent of the Toolchain Stack using the command `sam deploy --template-file template.yml --guided` from within the root directory of the repository.

The downside to a SAM CLI deployment is that it necessarily bypasses the IAM roles and policies used by the Toolchain to prevent privilege escalations. Thus, this deployment method is recommended only for development purposes, or in combination with a Toolchain-based deployment. The later case – deploying the Toolchain, but also using SAM CLI for manual development deployments – provides the permissions boundary (ensuring a compromised application cannot be used for privilege escalation). But, such a deployment still bypasses the CloudFormationRole, and therefore means a malicious SAM template could modify unwanted resources. This is assumed to a negligible risk, though, as a developer is assumed to be trusted with whatever permissions their account holds, and the SAM CLI displays proposed changes to the developer.

2.3.2. *Resources*

The application is foremost a Lambda Function, supported by the other resources in the Application Stack: three DynamoDB Tables, two SQS Queues, one SNS Topic, and a Pinpoint App with one toll-free phone number provisioned. The resources of the application, plus the user, are shown below in Figure 3.

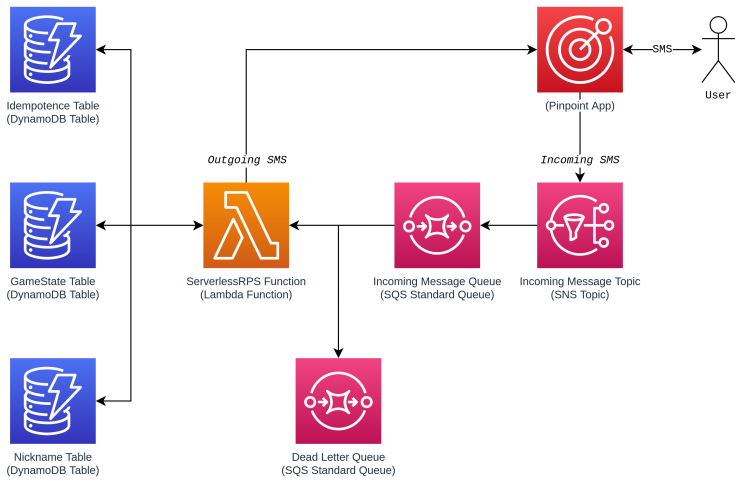

Fig. 3. Application Stack Resources

Players interact with the application over SMS. SMS messages sent to the toll-free number configured in Pinpoint are sent to the Incoming Message Topic. The Incoming Message Queue is subscribed to this topic, and will receive and queue incoming messages. The ServerlessRPS Function is triggered by the Incoming Message Queue, and will attempt to process batches of messages. Messages which fail to process are replaced in the Incoming Message Queue. Messages which repeatedly fail to process are moved to the Dead Letter Queue, for manual review. To provide idempotent message handling – as Standard SQS Queues provide *at-least-once* delivery (potentially invoking multiple Lambda Function instances) – the ServerlessRPS Function checks incoming message UUIDs against the Idempotence Table – rejecting messages, if a record exists, or creating a record and proceeding, if not. Records in the Idempotence Table are entered with a time-to-live and automatically deleted after expiration, to avoid resource waste.

The Nickname and GameState Tables are used for storing application state, and are therefore implementation details, but we will describe their purpose nonetheless. The GameState Table is indexed by player phone number, and stores the state of in-progress games with other players. A pessimistic, atomic test-and-set locking mechanism is used to avoid collisions at the level of individual player records. To ensure liveness, a release-and-retry scheme is used. The lock's granularity could be improved, by locking at the game level, rather than locking a player's entire state, but we assert that such an optimization would be meaningless, given the latency and throughput limits of SMS, and a human player's own throughput limits.

Finally, the Nickname Table is indexed by nickname, and simply stores the

phone number of the player behind the respective nickname. Nicknames are used for convenience, and privacy purposes – one need not share their phone number with another player, only their nickname. The Nickname Table exists to provide efficient lookups, without requiring a second, expensive index on the GameState Table. For convenience, and efficient reverse-lookups, a player's nickname is also denormalized onto their player record in the GameState Table.

3. Discussion and Conclusion

3.1. *Discussion*

3.1.1. *"Production Ready"*

We previously asserted that the Toolchain Stack is "production ready." Though significantly less featureful than "for enterprise" solutions, such as Warzon et al. [7], which automatically handle multiple test environments, separate production AWS accounts, and more, we argue the Toolchain described herein is nonetheless capable, especially for small teams, or individual developers.

Foremost, the Toolchain provides the repeatable, automated deployments expected of a modern workflow. Second, it provides the hardening necessary to ensure compromises (both of the development infrastructure (i.e., the GitHub repository), and the application (i.e., running Lambda instances)) cannot: A. be escalated, or provide lateral movement; or B. be used to provision unrelated resources (i.e., abusable EC2 instances). Finally, though not as convenient as an all-inclusive solution, the Toolchain can nonetheless be used for separate test and production environments. For example, one could setup a separate production AWS account, as described in Warzon et al. [7], and deploy the Toolchain to monitor some production branch of a repository. Separately, a developer may deploy – possibly many – test toolchains, or even simply develop using SAM CLI-based deployments to some test environment.

3.1.2. *A Liveness Edge-case: Lambda Timeouts*

A notable limitation of this work lies in a neglected edge case of the (un)locking scheme. Specifically, locks are ensured clear – regardless of exceptions, graceful return, etc. – by the use of Python's `finally` clause. Liveness under this scheme depends on the strong assumption that Lambda instances are not prematurely terminated, either by host failure, or timeout. This is a particularly poor assumption given the latter scenario: timeout. Currently, by default, Lambda functions have a three second timeout [2]. Anecdotally, SRPS's most complicated code path (the "throw" command, which involves the bulk of the game's logic and database interactions) has been observed to occasionally take slightly longer than three seconds – resulting in the premature termination of the instance.

Such a termination manifests in a subtle locking bug: a player who makes a throw is *sometimes* left with a stale lock. This occurs because the pessimistic locking

scheme releases the requestor's record lock at the very end of processing. Thus, if the function runs slightly past the timeout, and is therefore terminated before the unlock operation is performed, the requestor is left with a stale lock.

SRPS currently specifies an eight second timeout (see `template.yml` in the repository). Thus, this edge-case is unlikely to occur in a lightly used test environment. However, SRPS accepts multiple events in an invocation, increasing the likelihood of an incident in a more active production environment!

A simple timeout scheme is sufficient to workaround this edge-case. In fact, SRPS already includes a configurable `expiration_epoch_timestamp` field with its locks – by default, 10 seconds in the future from the epoch timestamp at which the lock is acquired. Currently, SRPS does not act upon this field. This "bug" has been intentionally left as a recommended starting point for the curious reader looking for somewhere to start hacking on this application. One candidate solution, which tests for expired timestamps using DynamoDB condition statements, has already been implemented for the idempotence scheme.

3.2. *Conclusion*

We provide and describe a "production ready" serverless toolchain, built on AWS CodePipeline, and deployed via AWS CloudFormation, which provides strong IAM-based security boundaries, and integrates with GitHub to provide continuous deployment of an AWS SAM application. To demonstrate our Toolchain, we provide, and briefly describe, an AWS SAM-based reference implementation of Rock-Paper-Scissors, played over SMS via AWS Pinpoint. This combination of features, and thorough description, represents coverage of a middle-ground neglected by both AWS SAM examples, and CloudFormation-based "quick start" solutions – the former being insufficient for production, and the latter a poor learning tool.

References

1. Update a github version 1 source action to a github version 2 source action. URL `https://docs.aws.amazon.com/codepipeline/latest/userguide/update-github-action-connections.html`. Last visited 2021-04-29.
2. Aws lambda developer guide, api reference, actions, createfunction. URL `https://docs.aws.amazon.com/lambda/latest/dg/API_CreateFunction.html`. Last visited 2021-04-29.
3. Aws pinpoint api reference. URL `https://docs.aws.amazon.com/pinpoint/latest/apireference/welcome.html`. Last visited 2021-04-29.
4. P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, Nov. 2019. ISSN 0001-0782. doi: 10.1145/3368454. URL `https://doi-org.proxybz.lib.montana.edu/10.1145/3368454`.
5. S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst,

C. L. Abad, and A. Iosup. A review of serverless use cases and their characteristics, 2021.

6. H. Shafiei, A. Khonsari, and P. Mousavi. Serverless computing: A survey of opportunities, challenges and applications, 2019.
7. A. Warzon, F. Brazeal, C. Guse, J. Yeras, and J. McConnell. Serverless ci/cd for the enterprise on the aws cloud, Feburary 2020. URL `https://aws-quickstart.s3.amazonaws.com/quickstart-trek10-serverless-enterprise-cicd/doc/serverless-cicd-for-the-enterprise-on-the-aws-cloud.pdf`.