Bringing Real-World Data Back to Development

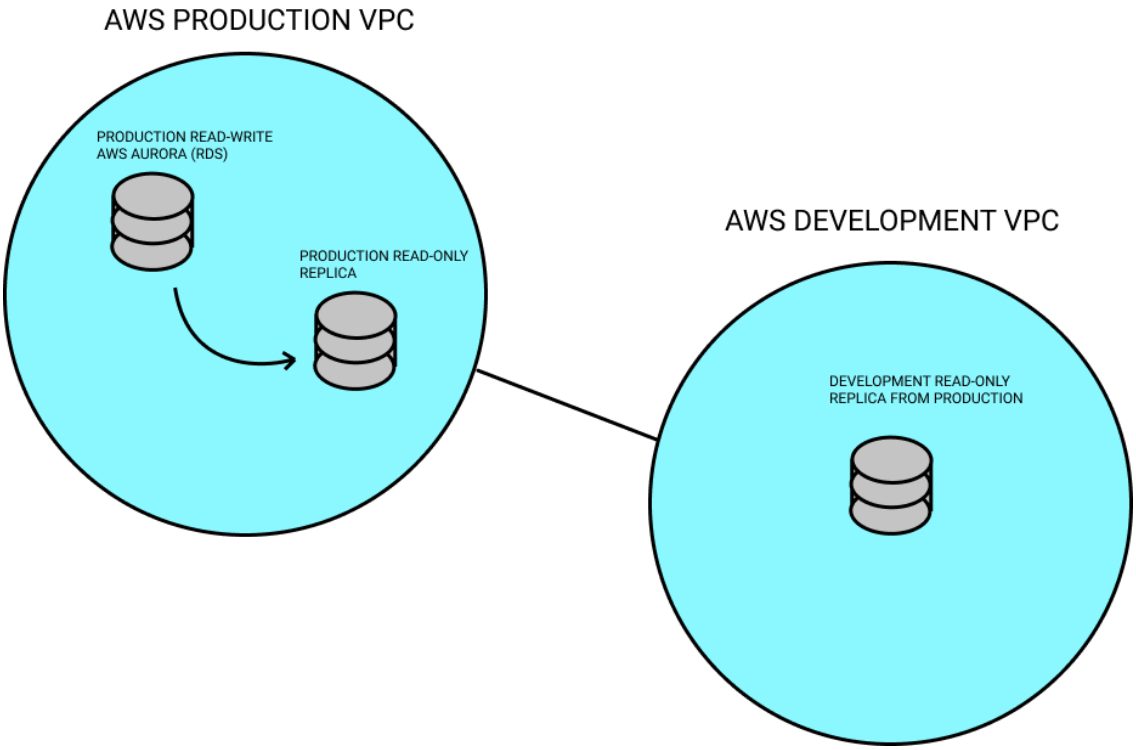
At Ignidus Technology, an Integrated Specialty Coverages company, we are continually striving to remove bottlenecks in our workflows and allow developers to maximize their time contributing to new features. Along with a variety of other initiatives, one way we do this is providing our development team with efficient access to real-world data, sanitized of course, to provide the closest possible replication of our production environment. This ensures developers are able to develop and test their changes in real-time against a production-like environment.

While many of the technologies in our process could be swapped for alternatives, for the purpose of providing context, here is a quick overview of applicable technologies utilized by Ignidus in our database refresh flow.

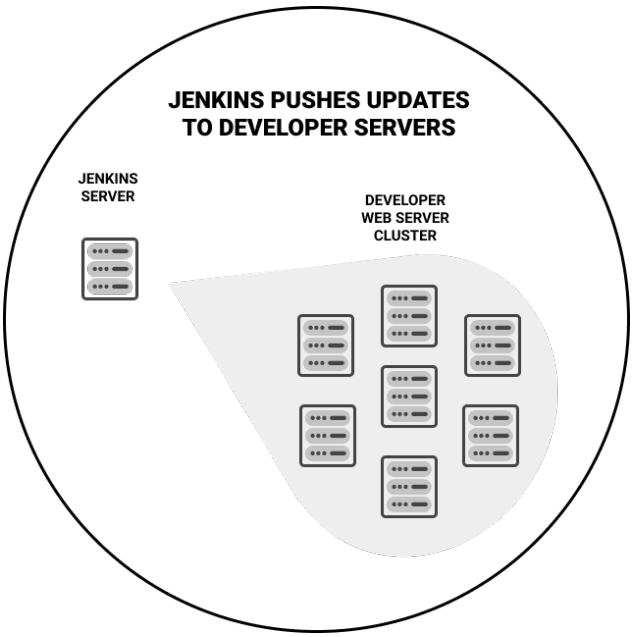
- AWS Virtual Private Cloud (VPC) with Peering
- AWS Aurora (MySQL and PostgreSQL-compatible solution by Amazon's Relational Database Service)
- Amazon Linux instances
- Jenkins Automation

Infrastructure Detail

To expand on the previous section, Ignidus utilizes Amazon's VPC for both our production and development server environments. While maintained under separate VPCs for security reasons, we have setup VPC peering between the two that allows us to run a production read-replica database in our development VPC. This is critical to our database refresh process as well as providing real-time read access to run urgent reports for our business.



For deployment and task automation, we utilize Jenkins, which has access via the AWS RDS API to create database snapshots as well as deploy changes to our development servers.



For security purposes, we've isolated our development VPC from our production environment with the exception of the VPC peering connection that allows for the setup of the prod read-replica in our development VPC. This isolation ensures that we do not have to open up access for SSH or other protocols to non-essential personnel or others, while still getting the benefit of having local (to the VPC) access to read the data and perform snapshots. Jenkins in our development environment is then granted special access through security group privileges to deploy these changes out to our development server cluster (ec2 instances). All SSH access to web resources as well as tools servers in both development and production require tunneling through respective bastion servers from whitelisted addresses.

While we've chosen to use AWS for our server infrastructure and Jenkins as our deployment tool, there are a number of other alternatives available based on organizational preferences if these are not right for your team. While it may present challenges with performance depending on throughput, you may even expand this to a cloud-to-on premise infrastructure or even both on-premise. Some compatible alternatives to AWS might be Google Cloud, Microsoft Azure, Digital Ocean or any number of other reputable cloud services. Jenkins as well has a number of alternatives that may fit your particular needs including, but not limited to, CircleCI and TravisCI.

Aurora Snapshots

In Jenkins, we have a project that triggers periodically with the following scheduled: 0 2 * * *. Each morning we run our dump PHP script to capture a snapshot of our production data from the read replica. This snapshot is the basis for every refresh run the following day, so it is absolutely critical that process runs consistently.

For the allowing us to quickly expand upon development needs in the system operations side at Ignidus, we have a composer-based microservice that brings in the following libraries:

```
"require": {
  "aws/aws-sdk-php": "^3.56",
  "zendframework/zend-console": "^2.7",
  "apache/log4php": "^2.3"
}
```

In our script that runs everyday at 2am, we allow for a –snapshot flag to be specified, captured by the script using the Zend\Console\Getopt class. In the script, we bootstrap the composer autoloader, setup logging, and then we are using the RdsClient SDK for AWS to run the *createDBClusterSnapshot* method on our Aurora cluster in order to create the base snapshot. It should be noted that we use a long-polling approach to determining if the snapshot has successfully created by calling the

describeDBClusterSnapshots method with the snapshot ID returned on initial call to *createDBClusterSnapshot*. Upon completion of the snapshot (or use of the latest system snapshot as a fallback), we restore the snapshot in order to begin dehydration.

Dehydration Overview

With the restoration of the snapshot, we are then able to use our process of "dehydrating" the database to get the data size down to a more efficient size, while still allowing for a breadth of usable data across critical tables for our development team to work with.

The initial step in the dehydration process is truncating unnecessary tables for development. This includes a number of historical tracking tables, logs, chats history as well as tables with the primary intention of storing mappings to file system documents that would not, for sensitivity purposes, be available in the development environments. This truncation is the initial chop of the axe, figuratively speaking.

The next step in the process centers around preservation of recent core records and all tables with which they have ties. Part of this process would be simplified by foreign key references with ON DELETE; however, at Ignidus, we have to be really careful to preserve write performance and as a result, do not use many explicit foreign keys. The relationships still exist in practice by having field references with IDs; however, we do not utilize MySQL / Aurora's foreign key concept in normal practice.

As mentioned, the key goal of this whole process is to preserve an assortment of recent data that allows the developers to have all of the critical data needs to optimize their productivity, but to balance that with an efficient refresh process that doesn't get bogged down or require excessive storage concerns on the individual developer MySQL databases stored locally on our development servers. For Ignidus, our most central component of data is the application for insurance. This maintains relationship references to producers and retail brokers, ratings (or pricing options), insurance products selected, documents generated and signed, etc. It is absolutely critical to our platform. Accordingly, we use this as our central point of data we need to maintain.

In order to do so, we maintain a certain minimum number of applications over a trailing period (for use, we maintain a little more than a year of applications with other filtering applied to ensure we have renewal applications to test with). Once the application list is filtered down and set, we then use the following SQL steps to remove unnecessary records from related tables:

```
CREATE TABLE temp LIKE base_table;
INSERT INTO temp (SELECT * FROM base_table WHERE app_id IN (SELECT app_id FROM
applications));
DROP TABLE base_table;
RENAME TABLE temp TO base_table;
```

The names / fields above are fictitious, but the concept is the same. SELECT records to preserve into a newly created temp table, DROP the current table, and RENAME the temp table to the current table name.

Upon completion of the dehydration process, we then use mysqldump to take a new minimized snapshot of the restored data for use in the refreshing developer databases. If an Aurora snapshot had been created earlier in the flow, this would be cleaned up in this phase as well, leaving only the dehydrated mysqldump files for the last 7 days available for developers to select from when refreshing. These files are then stored in our development VPC using Amazon's Elastic File System (EFS) with a mounted directory accessible from our Jenkins server.

Refresh Process

With the snapshots now available, we created another Jenkins project that all developers have access to run; configuration is limited to admin users only so as to not allow for accidental deletion or edits. In this project, developers or quality engineers can select from the various development servers for their target host as well as the copy of the snapshot to restore from of the available last 7 days. Additionally, we allow a selection of available application tags / branches from our integration with Git to support automatically run SQL migration updates available on the selected branch during the restore on the developer instance.

b. Restore minimized snapshot
c. Automated run of branch DB updates on refreshed database and Jenkins output logging

5) Summary / Benefits Discussion
a. engineering team benefits:
i. increased developer velocity since they are not having to spend time manually running SQL scripts / debugging data issues
ii. better developer visibility of how changes will impact production
iii. reduction in debugging time on production issues when production data can be brought back to dev environment
b. organizational benefits:
i. greater testability of code with comparable data
ii. improved release confidence
iii. shorter planning-to-release life cycle
c. recap of process / closing