

B. TECH. PROJECT REPORT

On

Optimizing operations involved in Cell Architecture

BY
Rohit Nikam



DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE

May 2022

Optimizing operations involved in Cell Architecture

A PROJECT REPORT

*Submitted in partial fulfillment of the
requirements for the award of the degrees*

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by:

Rohit Nikam

Guided by:

**Dr. Kapil Ahuja, Professor, IIT Indore, and Rohit Sharma, Engineering Manager,
Amazon**



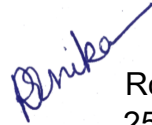
INDIAN INSTITUTE OF TECHNOLOGY INDORE

May 2022

CANDIDATE’S DECLARATION

We hereby declare that the project entitled “**Optimizing operations involved in Cell Architecture**” submitted in partial fulfillment for the award of the degree of Bachelor of Technology in ‘Computer Science and Engineering’ completed under the supervision of **Dr.Kapil Ahuja, Professor, IIT Indore and Rohit Sharma, Engineering Manager, Amazon** is an authentic work.

Further, I/we declare that I/we have not submitted this work for the award of any other degree elsewhere.



Rohit Nikam
25/05/2022

Signature and name of the student(s) with date

CERTIFICATE by BTP Guide(s)

It is certified that the above statement made by the students is correct to the best of my/our knowledge.



Prof. Kapil Ahuja, CSE, 25/05/2022

Signature of BTP Guide(s) with dates and their designation

Preface

This report on “Optimizing operations involved in Cell Architecture ” is prepared under the guidance of Dr.Kapil Ahuja, Professor, IIT Indore and Rohit Sharma, Engineering Manager, Amazon.

Through this report we have tried to give a detailed design of how one can reduce the ticket count by deduping the tickets and tried to cover every aspect of the design, if the design is technically and economically sound and feasible.

We have tried to the best of our abilities and knowledge to explain the content in a lucid manner. We have also added designs and figures to make it more illustrative.

Rohit Nikam

B.Tech. IV Year

Discipline of Computer Science and Engineering

IIT Indore

Acknowledgements

I wish to thank Dr.Kapil Ahuja, Professor, IIT Indore and Rohit Sharma, Engineering Manager, Amazon for their kind support and valuable guidance.

It is their help and support, due to which I became able to complete the design and technical report.

Without their support this report would not have been possible.

Rohit Nikam

B.Tech. IV Year

Discipline of Computer Science and Engineering

IIT Indore

Abstract

Department of Computer Science and Engineering

Bachelor of Technology

Optimizing operations involved in Cell Architecture

Teams use inbuilt-monitors to cut tickets whenever their service metrics are breaching thresholds. There are separate alarms for each service metric for each cell. Whenever there is a regional issue (like an AWS LSE (Large Scale Event)), teams get a lot of tickets for their cell services having the same root-cause.

Whenever there is a regional issue (like an AWS LSE (Large Scale Events)), teams get a multiple of tickets for their services with the same root-cause. This causes two problems:

1. During LSEs, it can lead to a lot of noise in the ticket queue and make focusing difficult for the on call.
2. All these tickets have to be closed manually. It is also possible to miss some tickets.

We need to reduce the number of tickets corresponding to the same issue. As the number of silos per region increases, this will become more important.

De-duping solutions should ideally have all these properties

1. We should reduce ticket count as much as possible
2. We should have one root cause per ticket
3. Implementation should be purely in terms of Carnaval (we'll avoid introducing DJS jobs/services/etc. to avoid new points of failure)

Table of Contents

Candidate's Declaration	5
Supervisor's Certificate	5
Preface	7
Acknowledgements	9
Abstract	11
Table of Contents	13
1 Introduction	15
1.1 Technology	17
1.2 Team	18
1.3 Problem Statement	19
2 Cell	20
2.1 Cell Architecture	21
2.2 Benefits	22
3 Solutions proposed	23
3.1 Pool-Based	23
3.2 Dependency Monitoring	25
3.3 Stage, Severity and Regional deduping (accepted)	27
3.4 Initiatives Taken	30
4 Supporting Tasks	31
4.1 Template designing	31
4.2 Load Testing	31

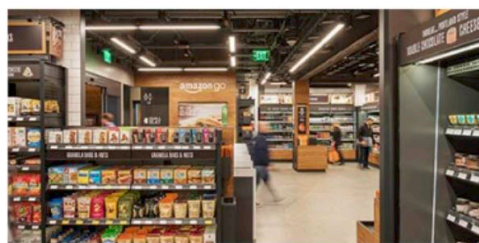
Chapter 1

Introduction

Amazon Go is a chain of convenience stores in the United States and the United Kingdom, operated by the online retailer Amazon. The stores are cashier less, thus partially automated, with customers able to purchase products without being checked out by a cashier or using a self-checkout station.

Amazon uses several technologies to automate Go stores, including computer vision, deep learning algorithms, and sensor fusion for the purchase, checkout, and payment steps associated with a retail transaction. The store concept is seen as a revolutionary model that relies on the prevalence of smartphones and geofencing technology to streamline the customer experience, as well as supply chain and inventory management.

Amazon Go is now available in two forms: Amazon Go Grocery and Amazon Go.



amazon go



amazon go grocery



Amazon Go offers delicious ready-to-eat breakfast, lunch, and snack options made by our chefs, favorite local kitchens and bakeries, and national brands. With our Just Walk Out Shopping experience, you'll never have to wait in line. No lines, no checkout. (No, seriously.)

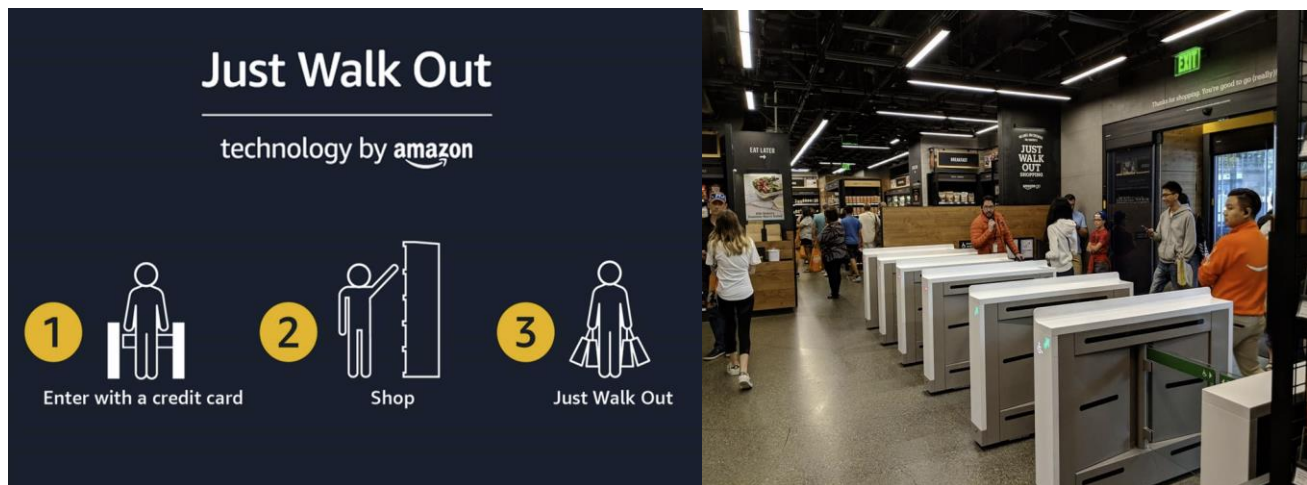
Amazon Go Grocery is the first grocery store to offer Just Walk Out Shopping—come in, take what you want, and just walk out. Amazon Go Grocery offers everything you'd want from a neighborhood grocery store—from fresh produce and meat and seafood to bakery items and household essentials—plus easy-to-make dinner options. We offer a mix of organic and conventional items from well-known brands, along with special finds and local favorites. With our Just Walk Out Shopping experience, you'll never have to wait in line. No lines, no checkout. (No, seriously.)

1.1 Technology

Just Walk Out Technology

The checkout-free shopping experience is made possible by the same types of technologies used in self-driving cars: computer vision, sensor fusion and deep learning. Just Walk Out Technology automatically detects when products are taken from or returned to the shelves and keeps track of them in a virtual basket.

With Just Walk Out Shopping, you can enter the store by scanning the In-Store Code in your Amazon app and scan the QR code, or hover your palm using Amazon One, or insert a credit/debit card. When you're done shopping, you can just leave the store. Later, we'll email you a receipt and charge your Amazon account. No queues, no checkout.



1.2 Team

Infrastructure Automation Team

Services expend developer effort for every new store bring-up be it in checking out LPT packages, synthesizing resources, validating endpoints, creating or enabling DJS Jobs etc. This effort increases multi-fold with the number of store launches planned this year and the number of stores that might come in future. To scale JWO further it is very important to offload Developers / Engineers from this manual effort.

We, Infrastructure Automation Team are providing a platform which automates Store Bring-up Tasks and aims to eliminate manual effort spent for maintaining of new Store in an existing region of stores. An Automated Job runs the registered Workflow on a daily basis. In case of failures a ticket is cut to the service owners which contains necessary information to debug the bring up failures.

1.3 Problem Statement

Situation

Teams use inbuilt-monitors to cut tickets whenever their service metrics are breaching thresholds. There are separate alarms for each service metric for each cell.

Whenever there is a regional issue (like an AWS LSE (Large Scale Event)), teams get a lot of tickets for their cell services having the same root-cause.

Effects

As we have lot of cells in a region:

During LSEs, it can lead to a lot of noise in the ticket queue and make focusing difficult for the on-call.

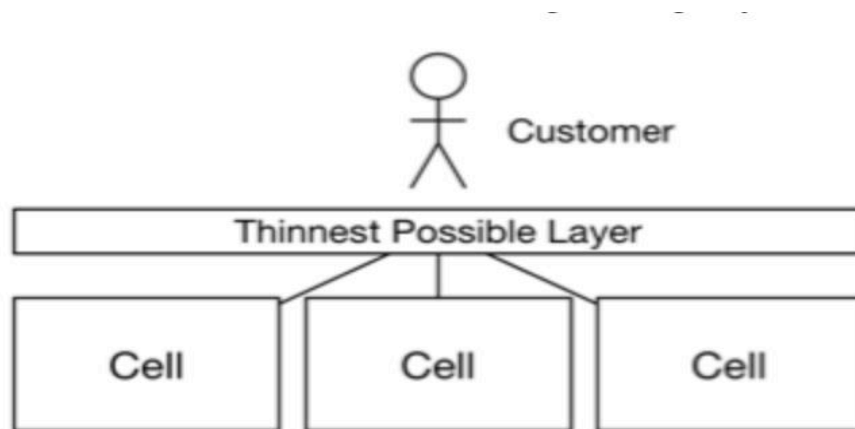
All these tickets have to be closed manually. It is also possible to miss some tickets.

These problems are becoming more painful as the number of cells per region is increasing.

Chapter 2

Cell

Rather than build out services as single-image systems, we propose a different approach: break our services down internally into cells and build thin layers to route traffic to the right cells.



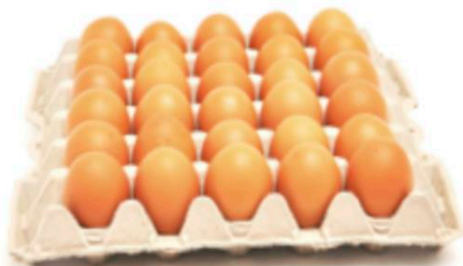
Each cell, a complete independent instance of the service, has a fixed maximum size. Beyond the size of a single cell, regions grow by adding more cells. This change in design doesn't change the customer experience of our services.

2.1 Cell Architecture

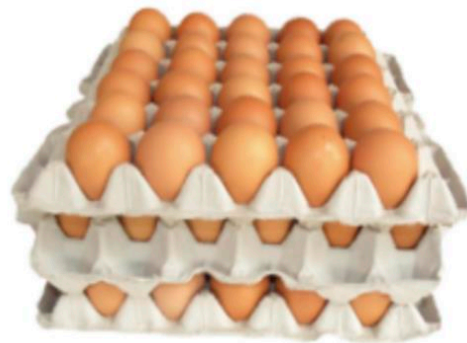
Cell architecture focuses on partitioning your service into small independent cells: rather than the traditional model of partitioning services by region, services are instead divided into smaller stacks, where each service stack receives a subset of traffic.

Cell architecture is next generation architecture pattern for highly available and scalable services. It divides service instance into smaller units along with data. All cells are completely isolated from each other. Services need to decide size of each cell.

Cell-based architectures benefit from capping the maximum size of a cell, and using consistent cell sizes across different installations (e.g., regions). Rather than using bigger cells in bigger installations, configure bigger installations to have more cells and smaller installations to have fewer cells, as depicted here:



Cell - 1



Cell - 2

2.2 Benefits

Cell-based architectures have a number of benefits:

- **Lower Blast Radius:** Breaking a service up into cells reduces blast radius. Cells represent bulkheaded units that provide containment for many common failure scenarios. When properly isolated from each other, cells have failure containment similar to what we see with regions. It is highly unlikely for a service outage to span multiple regions. It should be similarly unlikely for a service outage to span multiple cells.
- **Higher Scalability:** Cell-based architectures scale-out rather than scale-up, and are inherently more scalable.
- **Higher Testability:** The capped size of cells allows for well-understood and testable maximum scale behavior.
- **Higher mean time between failure (MTBF):** Not only is the blast radius of an outage reduced with cells, so should the probability of an outage. Cells have a consistent capped size that is regularly tested and operated, eliminating the “every day is a new adventure” dynamic.
- **Lower mean time to recovery (MTTR):** Cells are also easier to recover, because they limit the number of hosts that need to be analyzed and touched for problem diagnosis and the deployment of emergency code and configuration.
- **Higher Availability:** A natural conclusion is that cell-based architectures should have the same overall availability as monolithic systems, because a system with n cells will have n times as many failure events, but each with $1/n$ th of the impact. But the higher MTBF and lower MTTR afforded by cells means fewer shorter failures events per cell, and higher overall availability.
- **Safer Deployments:** Like one-box and single-AZ deployments, cells provide another dimension in which to phase deployments and reduce blast radius from problematic deployments. Further, the first cell deployed to in a phased cell deployment can be a canary cell with synthetic and other non-critical workloads, to further reduce the impact of a failed deployment.

Chapter 3

Solutions proposed

In total three solutions were proposed, out of which 1 got accepted.

- Pool-Based Approach
- Dependency Monitoring Approach
- Regional, State and Severity Deduping (Accepted proposal)

3.1 Pool-Based

The idea is that at a given time, there should not be multiple high severity tickets of the *same type* for the *same service* unless there's a regional issue going on.

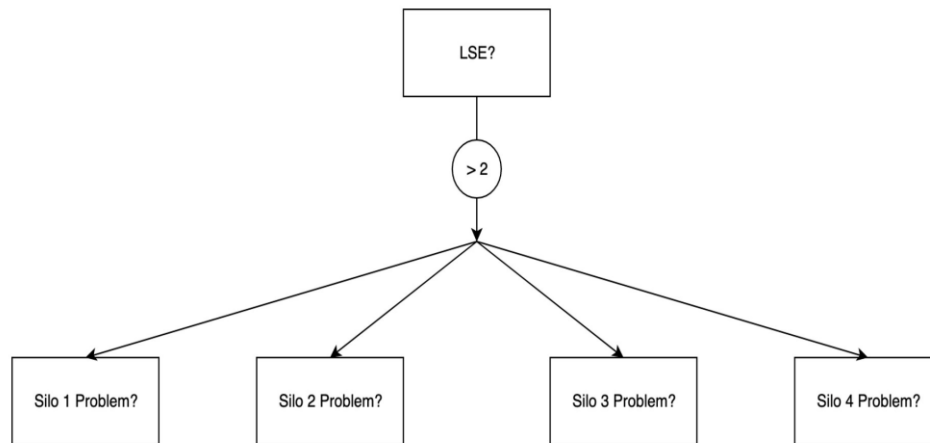
If more than one group of cells raises ticket for same issue then we check whether LSE (large scale event) is going on or not, if yes then no more similar tickets will get cut.

Pros

1. Makes the worst case better. In future, when there will be many more group of cells, a regional LSE could mean 10-15 high severity tickets for the same service. This will reduce that to 1 regional ticket + a few high severity ticket (that can be resolved in favor of the regional ticket).
2. We don't need to find out the dependencies. This makes it easier to adopt. Plus, it also handles cases that we didn't think of.
3. We don't need monitors for any dependencies. We don't have to worry whether we'll be able to find an aggregate monitor for a regional internal tool or not.

Cons

1. If the number of affected group of cells is low enough, we won't get a regional ticket.
2. We always get a few silo level high severity tickets and they have to be closed in favor of the regional ticket manually.
3. Requires educating the on-calls.
4. We double the number of monitors (earlier, there was only Problem? per group of cells; now there is the normal variant too).



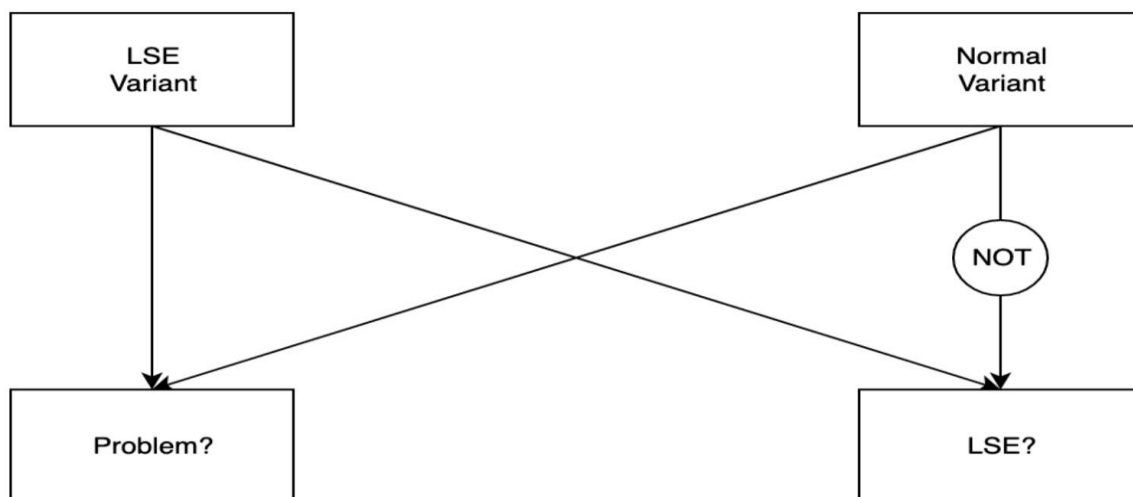
3.2 Dependency monitoring

We can *safely* enable regional de-dupe strings for very few ticket types as most ticket types are too broad (can be triggered due to multiple root-causes) or are rare enough that de-duping isn't worth it.

To increase the amount of regional de-duping, we can have 2 variants of broad monitors:

1. **LSE:** These cut tickets *when LSEs or regional issues are going on* and use regional de-dupe strings
2. **Normal:** These cut tickets during other times and use silo level de-dupe strings

To split a broad monitor with alarm rule Problem? into two, we create an alarm rule LSE? that becomes true during a LSE or a regional issue. Then the LSE variant's alarm rule becomes Problem? AND LSE? while the normal variant's alarm rule becomes Problem? AND NOT LSE?



Pros

- Allows regional de-duping for more ticket types

Cons

- If Problem? goes off before LSE? we'll not get regional de-duping and we'll get an additional ticket when LSE? goes into ALARM. This can happen as AWS monitors may go into ALARM after there are reports of issues.
- Potential LSE causes need to be identified for each service and have to be kept up-to-date.
- It might not be possible to find good monitors for all dependencies.
- We double the number of monitors (earlier, there was only Problem? per silo; now there is the normal variant too).

3.2 Stage, Severity and Regional deduping

Why stage deduping?

Code goes through various stages; we have shortlisted 2 consecutive stages which are very critical. The chances that these both stages will raise ticket of same root cause are very high.

Pros

- Stage-1 is meant to catch issues before Stage-2. So, the probability of Stage-1 and Stage-2 monitors going off due to the same reason is high.
- Will reduce the ticket count by n ($\text{Stage-1} \cap \text{Stage-2}$)
- Can be applied blindly. It doesn't matter if the monitors are high memory or high CPU.
- Minimal implementation effort required.

Why severity deduping?

Whenever we get a high severity ticket, there are often corresponding medium and low severity tickets. Due to their lower severity and earlier creation times, these tickets are often overlooked. This has 2 problems:

1. These tickets are often closed later without proper root cause analysis due to a lack of metrics/logs/data.
2. These tickets add clutter to the queue.

Pros

- This will bring reduce the number of tickets and clutter
- This will bring more attention to low and medium severity tickets
- Can be applied blindly most of the times. If the low severity de-dupes are very frequent and cut due to a lot more root causes than medium or high severity de-dupes, keeping them separate might reduce clutter in the de-dupe list.
- Minimal implementation effort required.

Cons

Low severity monitors can go off due to multiple reasons in a short amount of time and clutter the de-dupe list making it difficult to find high and medium severity de-dupes.

Why regional deduping?

Some types of tickets are only cut during a LSE and mostly have the same root cause.

We can avoid getting multiple silo-level tickets in such cases by using a regional de-dupe string.

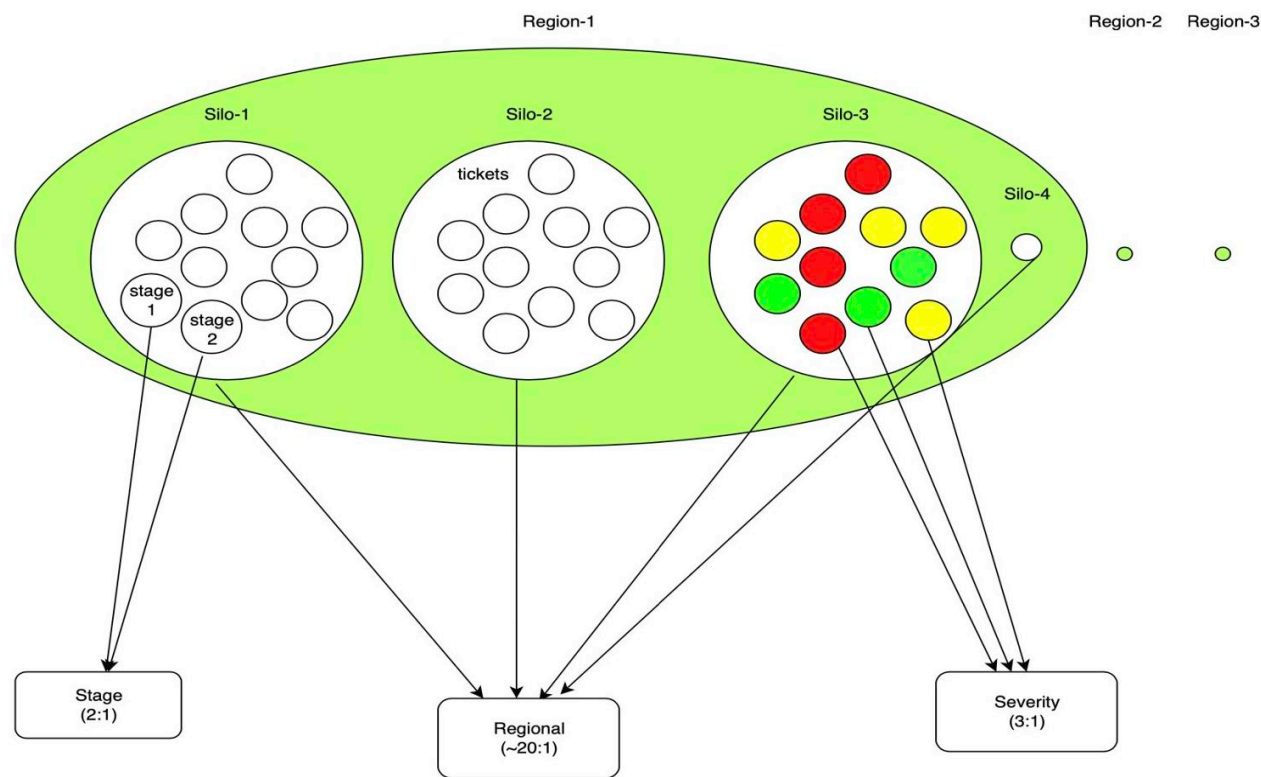
Pros

Makes things easier for the on-call during LSEs as they only have to look at one ticket per region

Cons

Highly specific to monitor type. We can only safely enable de-duping for Turtle credentials validity by default. Teams have to spend time to see whether they can enable de-duping for other monitors.

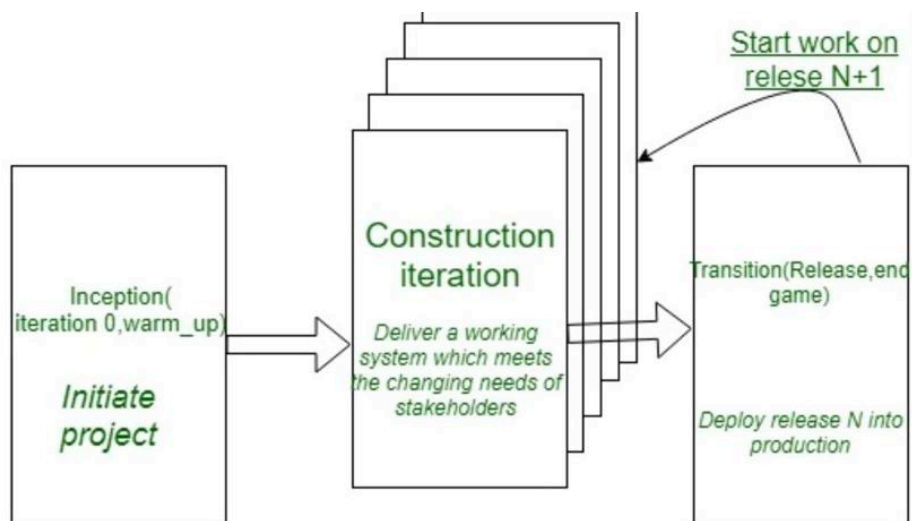
Stage, Severity and Regional deduping:



3.4 Initiatives Taken

Computer Science engineering principles applied

- Designed low level designs to fit the proposed deduping strategies. (To analyze the performance of the algorithm)
- Restructured and designed existing classes and data structures to incorporate deduping strategies. (To make efficient use of the resources and organize all the details of a cell.)
- Followed agile software principles to complete the entire project.



Agile software process three main strategies

Chapter 4

Supporting Tasks

These tasks were done along with the main project that is reducing ticket count, in order to make the main project successful.

4.1 Template Designing

Problem statement:

We have many config files corresponding to cells in a region and all of them have 80% same content, if some common change needs to be done, we end up editing all files (which are a lot).

Solution:

We created a template that generates all config files and if need to do some changes in all or few, we just modify the template and building it will generate all our config files with required changes

Actions Taken

Wrote a full-fledged template to generate the required configuration files. (Designed a ruby algorithm which spins on data and generates config files for all the cells)

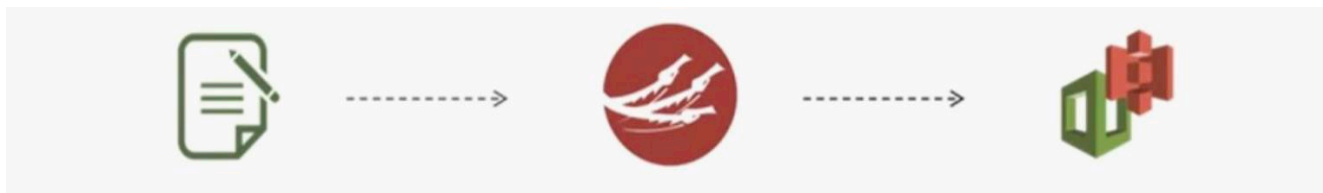
4.2 Load Testing

Problem statement:

As we have so many cells right now that the number of requests hitting the services might be overwhelming sometimes. This number becomes more significant as we grow with more cells per stores.

Solution:

Load Testing is a non-functional software testing process in which the performance of software application is tested under a specific expected load. The goal of Load Testing is to improve performance bottlenecks and to ensure stability and smooth functioning of software application before deployment.



Writing code to configure
test environment

Test environment configures
test infrastructure in the cloud

Essential components like S3,
lambda etc. gets setup



1. Test environment is triggered
after the deployment.
2. Test pass/fail status is
reported

1. Test environment orchestrates
the test run.
2. Receives and reports the
result.

1. Test cases are run against
the service
2. Logs are stored and
results are reported.

Initiatives Taken:

Created a dedicated Test Environment for load testing, determined load testing transactions for an application, Prepared Data for each transaction. Did test Scenario execution and monitoring on various metrics like global completion rate, global error rate etc. (Worked on cloud computing principles to do load testing in cloud, instead of doing it on local devices)

Chapter 5

Impact and Conclusion

After implementing the deduping strategies, teams(customer) using our services faced overall ticket cut count decreased by 25% after implementing stage deduping, by 35% after severity deduping and by 60% after implementing regional deduping.

Teams using our services are now getting only those tickets which are high priority, severe unique. Duplicacy and redundancy issues are also solved.