UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Sophio Japharidze

High Availability Deployments at Twilio Flex - a Case Study

Master's Thesis (30 ECTS)

Supervisor: Dietmar Pfahl

Tartu 2021

Dedication and Acknowledgements

This thesis is dedicated to the loving memory of my grandmother, Tsiuri. She has always been my motivation to keep pushing forward. This is for her.

I would like to thank my supervisor Dietmar Pfahl and all my coworkers at Twilio, who helped and supported me with this thesis. Special thanks to my Tech Lead, Teemu Suutari and my manager Margus Mahler for making it all happen.

Last but not least, I would like to thank my big family. They have always been there for me, believed in me, and, most importantly, made me believe in myself.

High Availability Deployments at Twilio Flex - a Case Study

Abstract:

It is important for Twilio Flex to minimize downtimes and the blast radius for breaking deployments to ensure that the Service Level Agreements (SLAs) are not violated, and a minimum number of customers are affected for a minimum amount of time. This paper has set a goal of implementing a Proof of Concept (PoC) of high availability deployments in Twilio Flex with this motivation.

In order to achieve this goal, a method was set out to select one of the Twilio Flex services for the PoC. Also, a suitable tool for high availability deployments was selected. Then, the PoC was designed, implemented, tested, and the results were evaluated.

The final evaluation shows that the PoC did not significantly affect the selected service's reliability, robustness, and speed. It was concluded that utilizing feature flags for gradual rollouts can bring benefits to high availability of Twilio Flex.

Keywords:

Twilio Flex, high availability, deployments, gradual rollouts, feature flag management

CERCS: P170

Kõrge kättesaadavusega kasutuselevõtt Twilio Flexis - juhtumianalüüs

Lühikokkuvõte:

Twilio flexi jaoks on oluline minimeerida aega, millal toode ei ole kättesaadav, ja vähendada mõjutatavate komponentide hulka, juhul kui uus versioon ei ole stabiilne. Selle eesmärgiks on pidada kinni SLA'dest ning minimeerida kasutajate hulka, keda katkine versioon mõjutab, ning perioodi, mille jooksul nad ei saa toodet korrektselt kasutada. Selle töö eesmärgiks on implementeerida Twilio Flexis eksperimentaalarendus kõrge kättesaadavusega tarkvara väljalasetest.

Selle eesmärgi saavutamiseks valiti kõrge kättesaadavuse tagamiseks sobiv tarkvara ning üks Twilio Flexi teenustest. Seejärel loodi eksperimentaalarendus, mida rakendati ja testiti ning saadud tulemustele anti hinnang.

Lõplik hindamine näitab, et PoC ei mõjutanud oluliselt valitud teenuse usaldusväärsust, töökindlust ja kiirust. Jõuti järeldusele, et funktsionaalsus lippude kasutamine järkjärguliseks tarkvar väljalaskeks võib Twilio Flexile kasu tuua.

Võtmesõnad:

Twilio Flex, kõrge kättesaadavus, tarkvara väljalase, järkjärguline tarkvara väljalase, funktsionaalsus lipu haldamine

CERCS: P170

Table of Contents

Terms and Notations

HVC - High Visibility Customer

SLA - Service Level Agreement

AWS - Amazon Web Services

VNV - Voice and Video

TTL - Time to Live

1 Introduction

Twilio Inc. is an American cloud communications platform as a service (CPaaS) company.

Twilio was founded in 2008 and is headquartered in San Francisco, CA, USA. Millions of developers every day use Twilio products in order to unlock the magic of communications and customer engagement. As its flagship and still leading products, Twilio offers developers communication channels like text, voice, chat, video, and email as APIs.

One of the younger Twilio products is Flex. Flex is a programmable, cloud-based contact center offering a wide range of channels, including Voice, SMS, WhatsApp, Facebook Messenger, and WebChat (Twilio, n.d.). This thesis will conclude a case study on high availability deployments in the scope of Twilio Flex.

Although Twilio Flex is still a young product, it has accumulated more than 600 customers two years after its launch in 2018. Revenue from Flex was up 184% in the first half of 2020. It is crucial to retain customer trust by reducing customer-facing incidents and service downtimes as much as possible. With more than 200 000 deployments a year, it is valuable to implement a deployment strategy that would minimize risks of negatively affecting customers, especially high visibility ones. High Visibility Customers (HVCs) in the scope of Twilio refer to customers that are strategically important to the company.

1.1 Motivation

In 2020, less than 1% of deployments at Twilio resulted in customer-facing incidents. However, in the first half of 2020 alone, more than 800 hours of related incident impact were observed. 70% of those hours were from just five incidents. These figures give a rough idea of how largely customers can be impacted even by a single service incident.

Twilio Flex, although initially created as a programmable contact center, has evolved into much more. Since the product can adjust to a customer's needs in unlimited ways, it is impossible to know all the ways the customers will use the product. For example, Twilio Flex is often used for emergency use cases, and consequently, any service disruptions could have a significant impact on the customers.

In addition to intangible losses (e.g., reputation damage) that come with service downtimes, monetary losses can also appear. If the Service Level Agreement, for example, the agreed monthly uptime percentage is violated, Twilio is obligated to compensate its customers as per their contractual agreements.

While service failures can never be fully eliminated, they can be redirected and managed. Minimizing the blast radius for failures would ensure that SLAs are not violated, and a minimum number of customers are affected for a minimum amount of time.

1.2 Goals

This thesis aims to implement a proof of concept of high availability deployments in Twilio Flex. The implementation will prove that it is possible to find and implement a deployment strategy that will help the company de-risk critical deployments and reduce negative impact for customers, especially high visibility and big ones.

In order to reach the goal, the best strategy for high availability deployments will be discovered. Afterward, a proof of concept (PoC) implementation will be done for the chosen strategy in an existing Twilio Flex service.

As a result of the PoC implementation, existing service figures (response times, load capacity, etc.) should stay unaffected. In addition, the PoC should bring additional benefits that were not present in the service before. It must be noted that the end goal within the scope does not include bringing the PoC changes into the production environment. Instead, the objective is to thoroughly test it in pre-production (DEV and STAGE) environments and assess its production readiness.

The thesis is structured as follows: Section 2 discusses the background and related concepts to high availability deployments inside and outside Twilio. It is important to note that the PoC implementation will only use the concepts and tools relevant to Twilio. Section 3 will describe the method and specific steps that need to be taken to achieve thesis goals. Then, Section 4 provides the results of each step defined in the previous section. Section 5 will describe lessons learned all the way through the method. Finally, Section 6 includes the conclusion and discussion about future work.

2 Background

Staying a competitive technology company in the modern world means delivering new features to customers in a continuous and timely manner. Promptly deploying changes to the production environment is an essential measure for a company's infrastructure.

This section will give an overview of the tools and concepts relevant to high availability deployments.

2.1 Relevant Concepts

Historically, different companies have been using different strategies to minimize downtime. Even individual teams in the same company might prefer various deployment methods, depending on their product and infrastructure.

Most deployment strategies aiming at high service availability involve introducing a change first to a small portion of the traffic or customers and increasing this portion if things go well. In other words, this process is referred to as a gradual feature rollout.

The following subsections will briefly describe some of the most common concepts related to gradual rollouts.

2.1.1 Feature Flags

Feature flags[1] is a simple technique (of course, there are more sophisticated applications to it, too) commonly used in practice. It is a basic if-else block around the feature that needs to be "guarded". Conditional statement inside the if block checks for the flag value to decide whether the enclosed code should be enabled or disabled. The value can be hardcoded in the codebase or set remotely.

As Mahdavi-Hezaveh et al. (2021) found out, 19 out of 20 surveyed companies use feature flags to have a gradual rollout. This means using flags to first introduce a new feature to a small group of customers and then, if successful, enable it for larger groups. It must be noted that in Twilio, feature flags are used to enable or disable a feature for *all* customers in the same Amazon Web Services (AWS) region (us-east-1, for example). To enable or disable features account-by-account, account flags are used. At Twilio, the synthesis of feature flags and account flags are called beta feature flags.

As simple and effective as feature flags sound, it is very easy to mismanage them and end up with substantial technical debt. It is essential to document, clean up and limit the use of feature flags since they add decision points to the code, which can be easy to forget. Knight Capital Group, an American global financial services firm, should know this the best. In 2012, they deployed a brand new feature, but it repurposed a feature flag from 8 years ago. Erroneous deployment caused 1 of 8 servers to run the dead code from 8 years ago. The engineers noticed that something was wrong 2 minutes after deployment, but they could not identify and fix it for 45 minutes. These 45 minutes cost the company $460 million, further followed by bankruptcy. (Seven, 2014)

Of course, feature flags mismanagement is only one of many reasons why the incident happened, but still, it is not a small one.

[1] Feature flags are also known as feature toggles, feature bits, feature flippers, and feature switches (Hodgson, 2017)

2.1.2 Blue-Green Deployments

Blue-Green[2] deployments can be considered a group of deployment strategies where the existing environment is first duplicated and then the load balancer directs traffic to the appropriate environment (Szulik, 2017).

This group of deployment strategies, first of all, includes the Blue-Green deployment strategy itself. This strategy is known as a "zero-downtime" one. The process involves duplicating the whole existing environment. Once the new environment is up and running and all the tests are passing, the entire traffic can be routed to the new environment. If necessary, rollback to the old version can also be done easily. This strategy is better suited for cloud infrastructure with container-based deployments and less suitable for data center-based ones (Rudrabhatla, 2020). Figure 1 shows the Blue-Green deployment strategy.

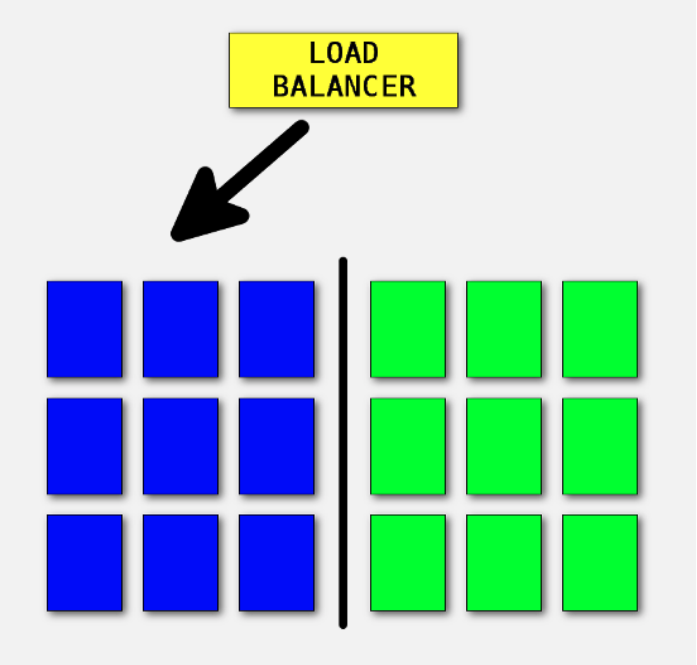

Figure 1. Blue-Green deployment (Szulik, 2017)

The second strategy in the Blue-Green deployment group is the canary deployment strategy. Canary deployments are more relevant to gradual rollouts as they involve deploying new versions of the application in small, incremental steps. Here too, the environment is replicated, but instead of switching the version all at once for all users, only some percentage of the traffic goes to the new node. Figure 2 illustrates this scenario. The percentage of traffic that goes to the latest version of the application can be increased gradually if the deployment is successful. The phased approach of canary deployment can be customized to the application needs. It can be implemented by switching the Blue version to Green to a certain user population based on the type of the user or the privilege they have (or) it can be based on the geo-location of the user (Rudrabhatla, 2020).

In the scope of Twilio, canary deployments refer to the customization of time. A single node with a new application version is booted up, and a fraction of traffic is routed to it. A time interval is specified after which the whole fleet switches to the latest version. For example, with a canary time of 60 minutes, first, a single node is booted up, and a fraction of traffic is routed to it. If for 60 minutes, there are no alerts or erroneous behavior from the new node,

[2] Blue-Green deployments are also known as Red-Black deploys (Budinsky, 2017)

the rest of the nodes are also booted up, and the whole traffic is routed to the new functionality.

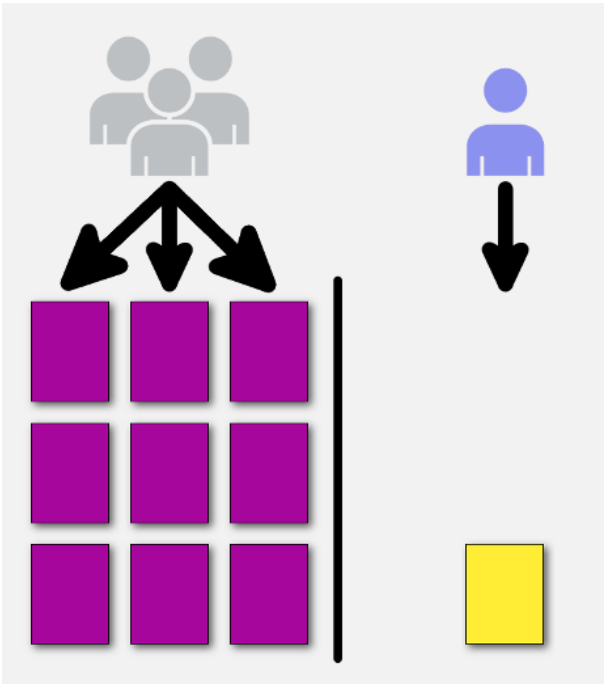

Figure 2. Canary Deployment (Szulik, 2017)

The last strategy in this group is the rolling deployment strategy. This process is about slowly replacing currently running instances of the application with newer ones (Szulik, 2017). Figure 3 illustrates the process of rolling deployments.

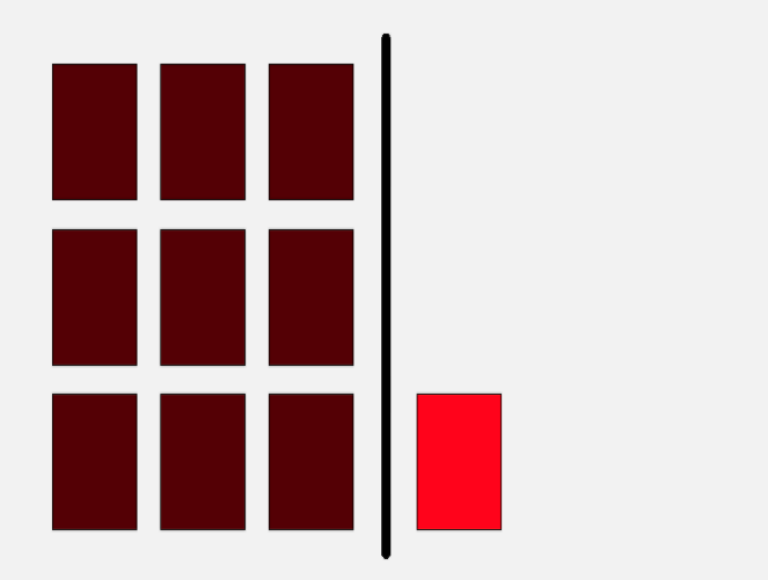

Figure 3. Rolling deployment

In this process, a new node is booted up, and once it is confirmed to be healthy, it is taken into the load balancer, and a single old node is taken out. Rolling deployments are usually faster than Blue-Green deployments. Still, since there is no apparent isolation between old

and new environments during rolling deployments, it is harder to roll back if the deployment fails (AWS, 2021).

Similar to rolling deployments is reverse rolling, which involves first taking the old host out of the load balancer and then putting the new host in.

2.1.3 A/B Testing

A/B testing[3] is a widely adopted technique by many large corporations like Amazon, Bing, Facebook, Google, Linkedin, and Yahoo! (Kohavi, Longbotham, 2017). Even though A/B tests are mainly used to experiment with different features, they can also be utilized for gradual rollouts.

A high-level structure of A/B tests can be seen in Figure 4. Treatment and control sets do not have to be split 50/50 (Kohavi, Longbotham, 2017). While traditional A/B tests can test how different colored buttons affect sales, with gradual rollouts its main goal is to observe the behaviour of application's new versions to reduce the risk of introducing breaking changes to the most important customer segment.

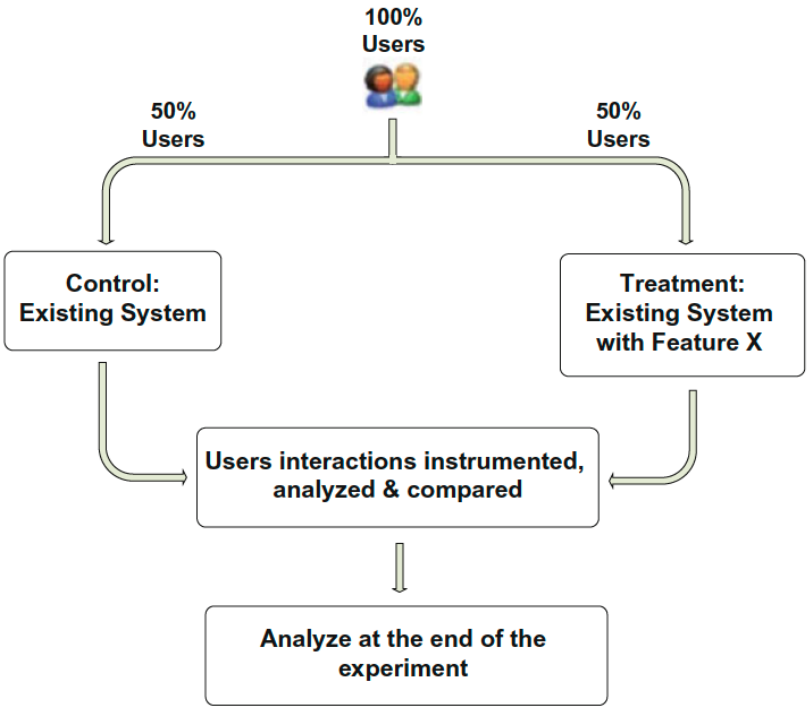

Figure 4. The high-level structure of A/B tests (Kohavi, Longbotham, 2017)

2.1.4 Other Concepts

It can be seen that the main idea of all the concepts and deployment strategies listed here is quite similar. In some cases, discussed concepts are strongly related. For example, feature flags can be seen as a means for canary deployments or A/B tests. The goal of this section was to simply introduce different existing notions. Even though other deployment strategies

[3] A/B tests are also called controlled experiments, split tests, randomized experiments, control/treatment tests, and online field experiments. (Kohavi, Longbotham, 2017)

exist such as recreate and shadow deployments, only those most relevant to gradual rollouts were presented separately.

Recreate strategy implies shutting down the old version and then booting up a new version (Tremel, 2017). Shadow deployment strategy consists of releasing the new version alongside the old version. The actual production requests are forked to the new version too, without impacting production traffic or response. This strategy is complex to implement and can be misleading as it is not a true user test (Tremel, 2017).

2.2 Deployment Tools and Practices

This section provides an overview of existing tooling and practices both inside and outside Twilio related to the deployment strategies and concepts presented in Section 2.1.

2.2.1 Feature Management Platforms

There are many commercially available tools for conducting gradual rollouts, mainly by utilizing feature flags. These tools are generally called feature delivery or feature management platforms.

Split is one of the feature flag management tools, but it also provides much more functionality than that. Split (n.d.) offers:

- Feature Flags
- Open source SDKs
- Management Console
- Data Ingestion & Export
- Monitoring & Alerting
- Experimentation & Impact Analysis
- Enterprise Security
- Workflow Integrations

Split offers the gradual rollout functionality to reduce critical deploy blast radius. Also, the tool provides a “kill switch” meaning that any unwanted new features can be turned off in seconds without rolling back the deployment.

Split makes it possible to synthesize feature flags logic with customer data to roll out a new feature gradually. In addition, they provide monitoring capabilities to observe the rollout status.

Although Split can handle various customer segmentation logics (different user populations, user attributes, or percentages), while the rollout is happening, it acts as a simple feature flag (or account flag in Twilio terms) for that account. Split’s feature flags are sticky. This means that all of the traffic for the affected treatment account is exposed to the new feature. Let’s say, for example, that the new feature is sending messages using a new API. With sticky feature flags, once the decision is made to expose an account to the new feature, all their traffic (i.e., all their messages) would use the new messaging API. Subsequently, if something were wrong with the new API, treatment customers would be hard down before the rollback would occur. An alternative for this would be to adjust the percentage of traffic for the percentage of customers exposed to the new feature. For example, 50% of messages of the mid-value customers get sent using the new version of the messaging API. This way, even if something were wrong in the new version, affected customers would not be forced to stop their business completely.

Alternatives to Split are companies like LaunchDarkly, CloudBees, Unleash, and many more. Although these companies all offer different pricing plans, the main functionality and features are more or less the same.

Like in Split, Launchdarkly's feature flags are also sticky. CloudBees percentage rollouts give the possibility to split rollouts by percentage, but again "This algorithm is similar to tossing a coin on the device. But once a coin is tossed, the result is consistent for a feature flag." (CloudBees, n.d.). Although it is possible to configure stickiness property - "the property that is the basis for stickiness behavior for a particular experiment." (CloudBees, n.d.)

On the other hand, Unleash gives the possibility to configure the activation strategy as "flexibleRollout" with the "stickiness" parameter set to "random". This means that there is "no stickiness guaranteed. For every isEnabled call, it will yield a random true/false based on the selected rollout percentage." (Unleash, n.d.)

2.2.2 Netflix

Netflix has its own platform called Spinnaker for continuous integration and delivery. Spinnaker is a sophisticated tool with many different functionalities that helps Netflix's systems be highly available.

To de-risk production deployments and limit the blast radius for breaking changes, they do regional deployments. This means that they deploy the service to one AWS region at a time and also try to avoid deploying during the peak hours for that region. (Glover & Probst, 2018).

Netflix also uses Blue-Green deployments (they call it Red/Black (Glover & Probst, 2018)) to be able to roll back the breaking changes quickly.

For canary deployments, they use the automated canary analysis platform Kayenta, which they have open-sourced in 2018. Netflix has slightly modified the canary release process, and they run three clusters instead of the traditional two in production. The first cluster is the old version of the application that has already been live for a certain period. The second cluster is called a baseline cluster, and it also serves the old version of the application but is newly booted up and typically contains three instances. Similarly, three instances are running in the third, canary cluster and serve the new version of the application. (Graff & Sanden, 2018)

Kayenta automates the process of deciding whether the canary cluster is safe and the process can continue, it is not doing well and needs to be rolled back, or if there is a need for manual intervention. The platform does this by first collecting the metrics from baseline and canary clusters and then judging. (Graff & Sanden, 2018)

2.2.3 Amazon

Amazon also uses canary deployments for their services to be highly available. Although it is common to define a time unit for testing the canary node, Amazon uses the number of requests as the unit. For example, start sending traffic to the canary node until 100 000 requests are reached. Then the decision can be made whether the canary node is doing well or not (Ramensky, 2019). This is a good method to calibrate deploying in non-peak hours (2 hours in non-peak hours results in fewer requests than during peak time).

Amazon has improved on the canary deployments in what they call fractional deployments (Ramensky, 2019). Traditionally, the percentage of traffic that goes to the canary node during a traditional canary deployment depends on the number of instances in the fleet (for example, in an 10 node fleet, 10% of the traffic goes to the canary node). Amazon has introduced additional logic in the load balancer that allows them to further adjust traffic percentage that

gets sent to the canary node. They can start as low as 1% and move up gradually up to 150% of the usual traffic to the canary node. (Ramensky, 2019). This allows testing the canary node under normal conditions, and load testing it for the cases if one of the other nodes goes down. After load testing, integration tests are run against the canary node, and these tests are called pre-production tests at Amazon. (Ramensky, 2019).

The fractional deployments process at amazon also includes automated anomaly detection. Their platform analyzes metrics generated by the canary node, and if the set threshold is exceeded, rollback is triggered.

2.2.4 Twilio

Twilio's CI/CD platform is called Admiral. Admiral is the central place for managing pipelines, deployments, and service configurations. The platform makes it possible to choose deployment (orchestration) strategy for the service, and the region where the deployment needs to happen. Figure 5 shows the list of deployment strategies currently available in Admiral.

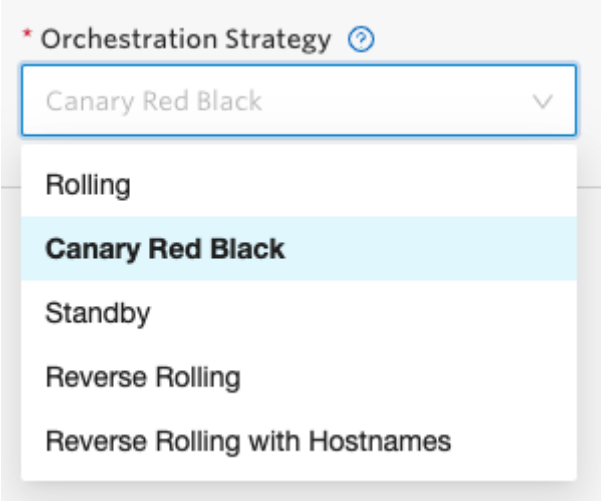

Figure 5. Deployment strategies at Twilio

Canary Red Black[4] brings one host from the formation (the 'canary') into the load balancer and monitors its behavior for a specified time. If the canary remains healthy, then the rest of the new hosts in the formation are brought into the load balancer all at once, and then the old hosts are removed. The period of time for testing the canary node can also be specified inside Admiral.

The Rolling strategy brings hosts into load balancer one by one (or optionally in small batches). After bringing in each new host, it takes an old host out of the load balancer. On the other hand, in the Reverse Rolling strategy, an old host is taken out of the load balancer before bringing in each new host. With Reverse Rolling with Hostnames, before bringing in each new host, an old host is taken out of the load balancer, the hostname is detached from it, and that hostname is attached to the new host.

The Standby strategy provisions hosts but does not bring them into the load balancer. Any hosts belonging to the formation that are currently in load balancer will be taken out of load balancer.

[4] Red-Black deploys are also known as Blue-Green deployments (Budinsky, 2017)

In day-to-day deployments, the Canary Red-Black strategy is most commonly used in Twilio Flex. For the most critical deployments, canary times can go as high as 24 hours. Although this reduces the blast radius of potential breaking changes, it is still not good enough because there is no way to know which customers will get served with the canary node. Let's say there are ten nodes in the fleet. If there is a fault in the new version of the application, only 10% of the traffic will get that change, but it is impossible to know which 10% of the customers will get it. Ideally, it should be guaranteed that the critical changes will not be introduced to high visibility customers and big customers in the initial phases of deployment. For this purpose, gradually rolling out changes to the customers is preferred. This way, customers are segmented based on their value to the business, and changes reach high visibility and big customers last, after they have been thoroughly tested.

Twilio has two main internal tools for managing gradual rollouts, which will be introduced in the following sections.

2.2.4.1 Gradual Rollout

Gradual Rollout is a Java library for feature flag management created by the Voice and Video (VNV) team at Twilio that enables engineers to safely and gradually roll customer traffic over to new behavior. The team created this library to solve problems that traditional deployment strategies had. As already mentioned, when a 10% canary is deployed, there was historically no way to control which 10% of traffic goes to the new path. This exposes all of the customers to change risk. For feature (account) flag migrations, one of the biggest problems with the process is account flags are all-or-nothing for a given account (flags are sticky). If a change is breaking for a customer, they will be hard down the minute they are rolled over. There has been no way to roll over only a portion of a single account's traffic, and the larger the account, the worse this problem becomes.

Gradual Rollout tackles both of the aforementioned problems. It provides a way to "partially" enable a code change for an account in such a way that a fraction of traffic is processed as if the change was enabled. The rest is processed as if the change was disabled which significantly reduces the impact for every customer when code changes are deployed. This behavior is the same as non-sticky feature (account) flags, allowing new features to be enabled for a fraction of traffic for a fraction of all accounts.

For monitoring rollouts, the team that is using this system is emitting Kafka events on each request. The events contain account and request information and the information about whether the feature was "on" or "off" for that specific request. The events are then streamed to ElasticSearch and can easily be visualized in Kibana.

Although some of the commercial tools could cover the functionality that the team was looking for, they are using this system for the critical path of requests, meaning that they would like to debug through each layer if/when needed. This is not so feasible while using third-party tools.

2.2.4.2 Geronimo

Geronimo is a Twilio internal stateless service built on top of Split.io. It provides an API for managing Split configurations. A handful of Twilio teams use Geronimo and, subsequently, Split.io to perform gradual rollouts by effectively managing feature flags.

Since Geronimo is just an API for managing Split.io, it does not provide any additional functionalities for gradual rollouts than Split does. It is possible to define customer segments in Split - e.g., high visibility customers, big customers, etc. It is also possible to make traffic allocation to a specific percentage, for example, 10%. This means that 10% of the traffic will

be evaluated against the experiment rules. So 10% of traffic gets compared to the segment lists to see if the accounts are allowed in. But once an account is evaluated as being "on" and the traffic allocation is the same or greater, that account will always be in the "on" group against the experiment. This is to say again that Split's feature flags are sticky.

As a final step, once 100% traffic is hit and there is a readiness to make the change permanent, the rollout can be "killed" by the default rule set, which in this case would be "on". So any time a split is called for this rollout, it will just respond with "on", and the code will act just as it did when the rollout was active.

For rollout monitoring, regular service metrics can be used. The dashboards and alerting need to be adjusted in advance to have clear visibility of which errors are coming from the old and new flows.

3 Method

This section describes the plan and the steps that are needed to implement gradual rollouts at Twilio Flex and evaluate its success. Figure 6 illustrates the method and its stages.

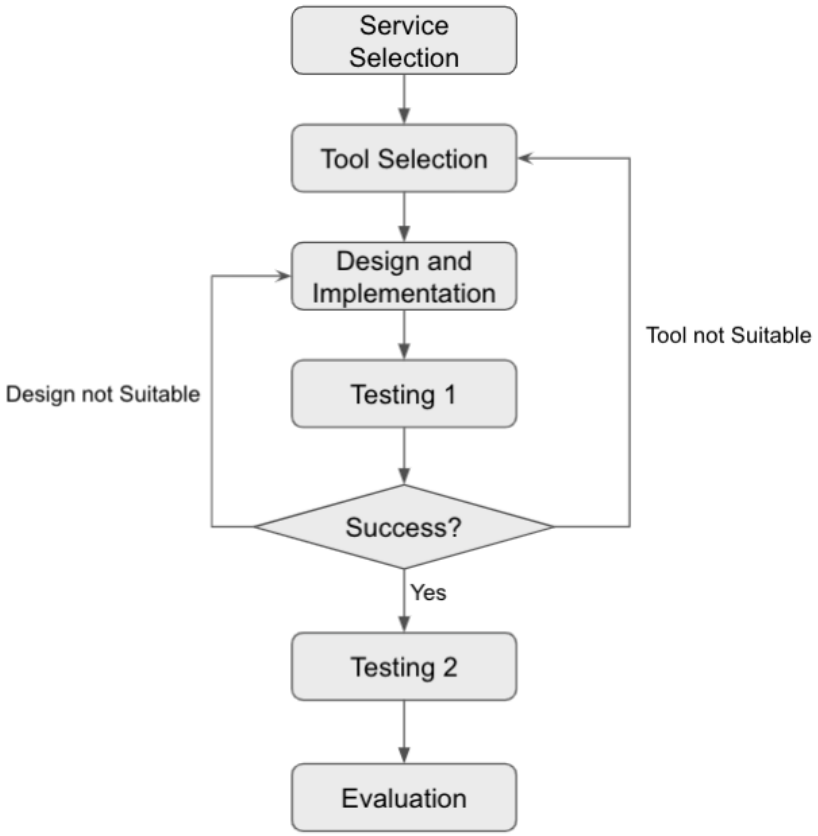

Figure 6. Method

The first item on the plan is to choose the service that will implement gradual rollouts. Next, a suitable tool needs to be selected. After the tool and the service are known, design and implementation can start. Two phases of testing will follow this step. Initial testing might conclude that either the design and implementation step failed or the selected tool is entirely unsuitable. Depending on the results, the process will go back to either the tool selection step or the design and implementation step. After the final testing, the whole proof of concept implementation will be evaluated to measure success. The following subsections explain in better detail what each step of the method includes.

3.1 Selection of the Service

To best illustrate the results and impact of the work done, first, a service that would benefit the most from the gradual rollouts needs to be chosen. The selected service needs to be of critical importance for the whole Twilio Flex and should have strict SLA defined. This is important because such services gain the most significant advantage from gradual rollouts, and potential positive impact can be better measured. Also, it is better if the service is being worked on and deployed regularly. This means that the code is being updated with new features, bug fixes, and security fixes regularly. “Regularly” here can be defined as at least one merged Pull Request per month. Gradual rollouts in such a service make more sense.

For analytics purposes, it would be beneficial to have clearly defined quantitative metrics in advance, such as what is the average response time for the service? How much load can it withstand? What is the average load on it in production? Such metrics will be useful in choosing a proper tool and for baseline purposes for final evaluation.

To summarise, suitable service:

- Is critical to Twilio Flex
- Has SLA defined
- Has frequent code updates
- Has defined metrics
- Is load tested

Technical leads from Twilio Flex will be interviewed to make service selections. To summarize, the end goal of this step is to select a service where the gradual rollouts proof of concept will be implemented.

3.2 Selection of the Tool for Gradual Rollouts

To make a proof of concept for gradual rollouts at Twilio Flex, a suitable feature flag management tool needs to be selected. Selecting an appropriate tool for gradual rollout can be considered even more important than further integration, testing, and evaluation, as all further steps depend on the first one.

To make selection objective and reproducible, selection criteria need to be defined:

- **Tried and tested**. It is important that the tool has been in use by at least one team in Twilio and is proved feasible to integrate it with Twilio services. Thus, candidate tools for the selection are those described in Section 2.2.4.
- **Reliability**. The chosen tool needs to be reliable. It needs to handle sufficient load so that the tool is not the reason for the service failure. Based on the queries run on Twilio Flex services in production, the average maximum load was 929 requests per second. The chosen tool needs to be able to handle a load higher than that, at least 1000req/sec.
- **Speed**. Similarly, the tool needs to be fast. The check of whether to expose the new feature to the customer or not cannot take more time than the actual request. According to the queries run in production on Twilio Flex services, the minimum response time was 5 milliseconds. Thus, the tool needs to have response times in this range to not introduce timeout errors for the end-users and the dependent services.
- **Easy to manage**. It is also important that there is a defined way to check rollout metrics for the tool. The team needs to be able to monitor rollout status in real-time and receive alerts in case something goes wrong. Furthermore, should the rollout go wrong, it should be possible to quickly roll back either fully or partially. Similarly, a "nice to have" feature would be easily manageable "roll forward", increasing rollout percentages.
- **Language-agnostic**. Since most of the backend services in Twilio Flex are written in Java, the tool needs to be compatible with it. Ideally, the tool is language-agnostic, enabling the teams to integrate it with any desired service.
- **Support**. Finally, it is important that the tool comes with relevant support. This means that there are people ready and dedicated to assisting the team with setting up the integration, defining, and managing rollouts and metrics.

To make the selection, teams using candidate feature flag management tools will be interviewed to discover how the tools meet the specified criteria. For the quantitative

analysis, such as load and speed requirements, specific examples and dashboards can be analyzed from those teams.

To summarize, the end goal for this step is to select a gradual rollouts tool that will be used for the proof of concept implementation.

3.3 Design and Implementation

As a third step, tool and service integration needs to be designed and implemented. For the design part, sequence diagrams will be created to describe service logic and what part the chosen tool plays in it.

After the designs are done, the tool needs to be integrated with a selected service. Specific implementation sub-steps will vary depending on the service and the tool chosen in the previous two points. This step also includes necessary unit and integration testing.

After the initial stages of implementation and integration are completed, metrics need to be configured. Depending on the tool, necessary metrics may be already provided out of the box. In other cases, it might be needed to modify traditional service metrics or dashboards or even modify the service so that it emits Kafka events on each request.

End goal of this step is to have deployment-ready, proof of concept implementation for the gradual rollouts.

3.4 Testing 1

After the metrics are ready, the service can be deployed to the DEV environment. Before that, baseline metrics need to be defined to see whether the process is successful or not clearly. In addition, to make a successful DEV deployment, all existing integration tests for the service need to pass.

Once in DEV, load tests will be performed, and the resulting metrics will be compared against the baseline metrics. If the load tests are successful, e.g. the integration can withstand the load, the next step can be performed. If different implementation iterations still cannot withstand sufficient load, the chosen tool can be considered to have failed the use case.

This step only includes running automated tests as opposed to also testing features manually. Since the DEV environment is in constant development, it is not representative of the real, stable behaviour. Hence, manual testing results would be skewed.

The goal of this step is to identify whether the proof of concept implementation is ready to go to the STAGE environment.

3.5 Testing 2

If all tests conducted in the DEV environment are successful, the service can be promoted to the STAGE environment. In Twilio, this is a stable, pre-production environment. Before deployment, test accounts will be created and segmented as artificial “regular customers”, “big customers”, and “high visibility customers”. Upon STAGE deployment, service cluster tests are automatically run, and the pipeline will roll back if the tests fail. If the feature gets to STAGE, additional tests will be performed to verify that the new feature is being deployed as expected.

This step is the final destination for the proof of concept implementation in the scope of this thesis. The goal of this phase is to evaluate the production readiness of the gradual rollouts tool-service integration.

3.6 Evaluation

The final step is evaluating the whole past process. In order to objectively assess results, success criteria need to be defined. Integration with the gradual rollouts tool can be deemed successful if:

- **Service performance is unaffected**. If there is no significant change in average response time from the service.
- **Integration is reliable**. If existing application load tests are passing after the integration with the gradual rollouts tool.
- **Integration is robust**. If gradual rollouts outage does not cause service outage/incident.
- **Rollout results are transparent**. If the rollouts can be monitored live and metrics can be filtered based on treatment class.
- **Integration is beneficial.**

It is important to note that the evaluation will be done based on DEV and STAGE deployments only. If the result of the evaluation shows success, it means that the integration is ready for the production environment.

In order to evaluate the last success criterion (whether the integration is beneficial or not), a subject matter expert will be interviewed. The interviews will be subjective and will show whether the PoC integration will bring benefits to the current deployment processes or not.

4 Results

This section discusses how the steps outlined in the method from Section 3 were actually carried out. The structure of this section will follow the steps outlined in Figure 6 (Method), summarising the results of each of the mentioned steps.

4.1 Selection of the Service

The technical lead at Twilio Flex was interviewed to identify the service where PoC for gradual rollouts will be implemented. During the unstructured interview, all points highlighted in Section 3.1 were discussed. The three most critical services for Twilio Flex were shortlisted. Candidates were Configuration Service, Provisioning Service, and Channel Orchestration Service. They all have strict SLAs and are depended on by numerous other Flex services. Also, they have defined baseline metrics for response time and load capacity.

The difference between the services is their purpose. Configuration Service is a central place where all Twilio Flex account configurations are created, deleted, updated, and retrieved. So, in essence, it is an API that offers configuration CRUD operations. If Configuration Service were down, the whole Flex would be down as well. That said, there is not much complex business logic involved in its implementation. The most common updates of this service involve configuration schema changes as opposed to the actual code changes. Figure 7 shows Github repository commits frequency per week for a one year period for the Configuration Service.

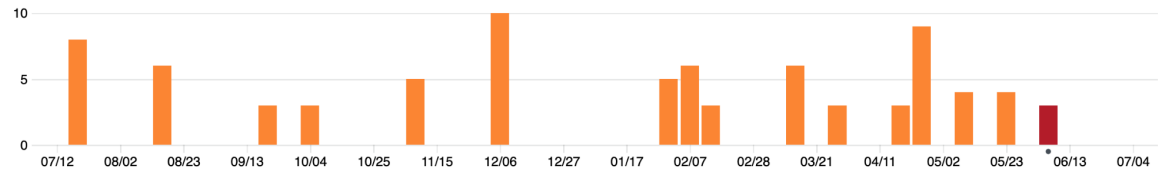

Figure 7. Number of commits per week for Configuration Service

From Figure 7 it can be calculated that the Configuration Service has an average of 1.5 commits per week. In addition, Configuration Service repository insights show that the service has had one merged Pull Request during a one month period.

Provisioning Service is responsible for creating Flex agents and overall user management in Flex. This includes third-party Single Sign-On (SSO) integrations to allow users to log into Flex. If the Provisioning Service were down, contact center agents would not be able to log into Flex. Figure 8 shows commits frequency for Provisioning Service. It can be calculated that the Provisioning Service has an average of 0.65 commits per week. In addition, repository insights show that Provisioning Service has had one Pull Request during a one month period, but this Pull Request has not been merged.

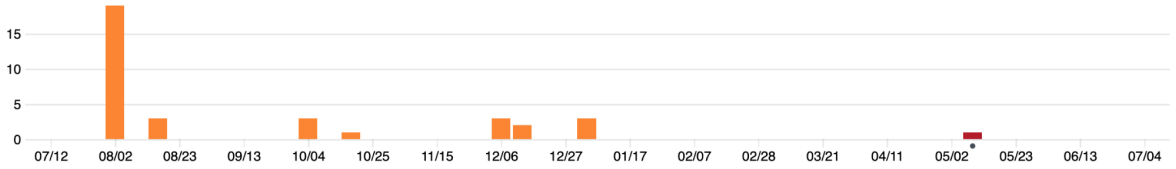

Figure 8. Number of commits per week for Provisioning Service

Channel Orchestration is a complex service responsible for the creation of Chat Channels and Chat Identities related to Twilio Web Chat and Twilio Channel Message use cases, Configuration of Channels (Facebook, SMS, etc.) for Flex messaging, and Janitor for the created Channels as well as Task wrap up. If Channel Orchestration were down, new chats would not be started, and existing ones would not be properly completed. Channel Orchestration is more prone to critical changes that would benefit from gradual rollouts. Figure 9 shows the commits frequency for Channel Orchestration. It can be calculated that the Channel Orchestration Service has an average of 2.2 commits per week. In addition, the repository has had 4 Pull Requests during a one month period, and 3 of them were merged.

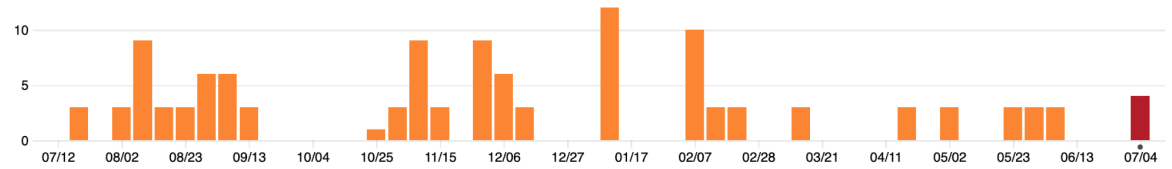

Figure 9. Number of commits per week for Channel Orchestration

Table 1 summarizes the service selection results.

	Configuration Service	Provisioning Service	Channel Orchestration
Critical	Yes	Yes	Yes
SLA defined	Yes	Yes	Yes
Load Tested	Yes	Yes	Yes
A lot of components	No	Yes	Yes
Frequently worked on (implementation changes)	No	No	Yes

Table 1. Summary of services' compliance with the criteria

Table 1 shows that all three services are equal with the first three criteria. The difference comes in with the last two - code complexity (a lot of components) and change frequency. The last criterion turned out to be the decision-maker. As previously discussed, Provisioning Service has 0.65 commits per week on average, while Configuration Service has 1.5 and Channel Orchestration has 2.22. While both Channel Orchestration and Configuration Service satisfy the criterion of at least one merged Pull Request per month, Channel Orchestration Service has more merged Pull Requests as well as a higher number of commits per week. It can be concluded that the Channel Orchestration is the best-suited service for the PoC of gradual rollouts.

4.2 Selection of the Tool for Gradual Rollouts

Table 2 shows the summary of conducted interviews with the teams, discussing the gradual rollout tools' functionalities and how they meet the criteria listed out in Section 3.2.

	VNV Gradual Rollouts	Geronimo (Split)
Tried and Tested	Yes	Yes
Reliability	500 req/sec per host and easily scalable;	550 req/sec per host;
Caching	Caching on both client and server-side	No caching pre-implemented; can be done by client;
Fast	99% under 25ms	99% under 10ms
Rollout Management	Manage rollouts directly using API;	Manage rollouts directly using Split Console;
Metrics	Produce Kafka events using ready-made endpoint; Manage metrics in Kibana (ElasticSearch)	Publish regular service metrics and monitor in Datadog
Language-agnostic	Yes (It's an API)	Yes (It's an API)
Support	No dedicated team	Yes

Table 2. Summary of gradual rollout tools' compliance with the criteria

Some of the criteria defined in Section 3.2 have been split into more granular ones for the differences between the tools to be better understood. It needs to be mentioned that the color coding (green and yellow) of Table 2 is relative and not absolute, meaning that the color yellow simply means that green is a more optimal choice for the current scope.

Metrics in Table 2 refer to how tools allow publishing data about how the service is performing. Channel Orchestration already has a defined set of metrics that are being published and monitored in real-time in Datadog. With Geronimo, it is possible to naturally evolve existing metrics and monitor them in the same (Datadog) environment. For VNV Gradual Rollouts, the recommended approach is publishing Kafka events using a ready-made endpoint every time a request is made to the service and then visualize these metrics in Kibana. Geronimo's approach is better suited for the current scope as for an engineer making deployment it is easier to monitor all the metrics in a centralized place.

Geronimo is owned by the experimentation platform team, whose purpose is to offer experimentation tools to other teams. Supporting teams who use their services is one of their main priorities. While Voice team engineers are enthusiastic about supporting other teams who will use the VNV Gradual Rollouts tool, they might not have the capacity to do so. It is also worth mentioning that VNV Gradual Rollouts was designed by Voice team engineers for their own use case and specific purpose, while Geronimo is a more general-purpose tool with many possible use cases.

To summarize, Geronimo (Twilio internal tool built on top of Split.io) is the selected tool for gradual rollouts for the PoC implementation.

4.3 Design and Implementation

This chapter describes how the PoC of gradual rollouts in Twilio Flex is designed and implemented, as planned in Section 3.3.

4.3.1 Design

This subsection describes the context, or the specific use case for which the gradual rollouts PoC will be implemented. It also briefly introduces a high-level description of the building blocks - the service and the tool for the PoC. Finally, this section defines and explains the specifics of the design for the PoC implementation.

4.3.1.1 Context

The current implementation of Channel Orchestration is fundamentally in the same state that it was on Flex launch in 2018, apart from the required ongoing fixes and some spot features. The intention was that Channel Orchestration would be a temporary service before moving into a new model, rather than a permanent solution. However, it has become apparent that the current solution cannot be deactivated any time soon.

The service uses a dedicated MySQL cluster to store its state. Currently, one of the biggest operational issues with Channel Orchestration is that the performance of MySQL queries is degrading with time. This is partially related to the fact that the database is constantly growing. In addition, the data that is only needed temporarily is currently also stored in MySQL and not cleaned up later. More specifically, there is a row created in the `chat_users` table every time a new Web Chat Channel is created (a message comes into the contact center from a webchat). The "user" object is only needed for chat creation and the data is safe to remove afterward.

One of the short-term remediations for the operational issues would be to move away this temporary users' data from MySQL to DynamoDB. DynamoDB allows defining a Time to Live (TTL) attribute and the inserted rows can be automatically retired after the specified time.

In practice, implementing this remediation means redirecting all existing MySQL writes and reads for a specific table to the new table in DynamoDB. This is a big change since if the creation of the row in the `chat_users` table fails, Web Chat Channel creation will fail. This will negatively affect a lot of the customer's traffic and Twilio Flex's SLAs. In order to make the initial deployment successful, it would make sense to employ gradual rollouts. Hence, MySQL - DynamoDB migration for the `chat_users` table is the PoC feature for which the gradual rollout functionality will be implemented inside the service.

4.3.1.2 Building Blocks

Before going into the details of the actual PoC design, it is important to get an overview of its building blocks - the tool and the service. As discussed in Sections 4.1 and 4.2 respectively, the chosen service for the PoC is Channel Orchestration and the chosen tool is Geronimo.

Geronimo is an experimentation API and it is a stateless service. Geronimo uses SplitSDK to evaluate the subject for the experiment. No network request is made, no data leaves Twilio. All requests to Geronimo are published to Kafka and later streamed to Presto.

As already mentioned in Section 4.1, Channel Orchestration is one of the critical services for the whole Twilio Flex. It handles the creation of Chat Channels and Chat Identities related to Twilio Web Chat and Twilio Channel Message use cases, Configuration of Channels (Facebook, SMS, etc.) for Flex messaging, and Janitor for the created Channels as well as

Task wrap up. Its criticality means that not only the service cannot have any significant downtime, but it also has to be reliable and fast.

4.3.1.3 Specifics

As discussed in Section 4.3.1.1, the PoC feature is MySQL - DynamoDB migration of the `chat_users` table. This essentially means that whenever a request for Web Chat Channel creation comes in, a query will be made to Geronimo. If the account has the new feature flag "on", a row for the `chat_user` will be created in the new DynamoDB table. In any other case, the service will act as before, creating a row in the existing MySQL table.

Current SLA for Channel Orchestration defines that P99 of the requests need to be completed in under 500 milliseconds. This means that all possible wastes in execution time need to be minimized. This includes waiting for the responses from the upstream services.

Since Geronimo SLA defines that P99 of the requests will complete in under 10ms, the PoC can be designed so that Channel Orchestration does not wait for Geronimo's response longer than that. An example sequence diagram of this design is shown in Figure 10.

In the example from Figure 10, a request is made from Flex UI to the Channel Orchestration Service to create a Web Chat Channel. With the gradual rollouts behavior, in addition to the existing logic for creating a Web Chat Channel, Geronimo Service will be called to get treatment for the account. If Geronimo takes more than 10ms to respond with a variation ("on" or "off") for the account, the service will not wait anymore and automatically assume the variation to be "off", meaning that the service will act as it has been before. It should be noted that this request for webchannel creation is one of many use cases of the Channel Orchestration Service.

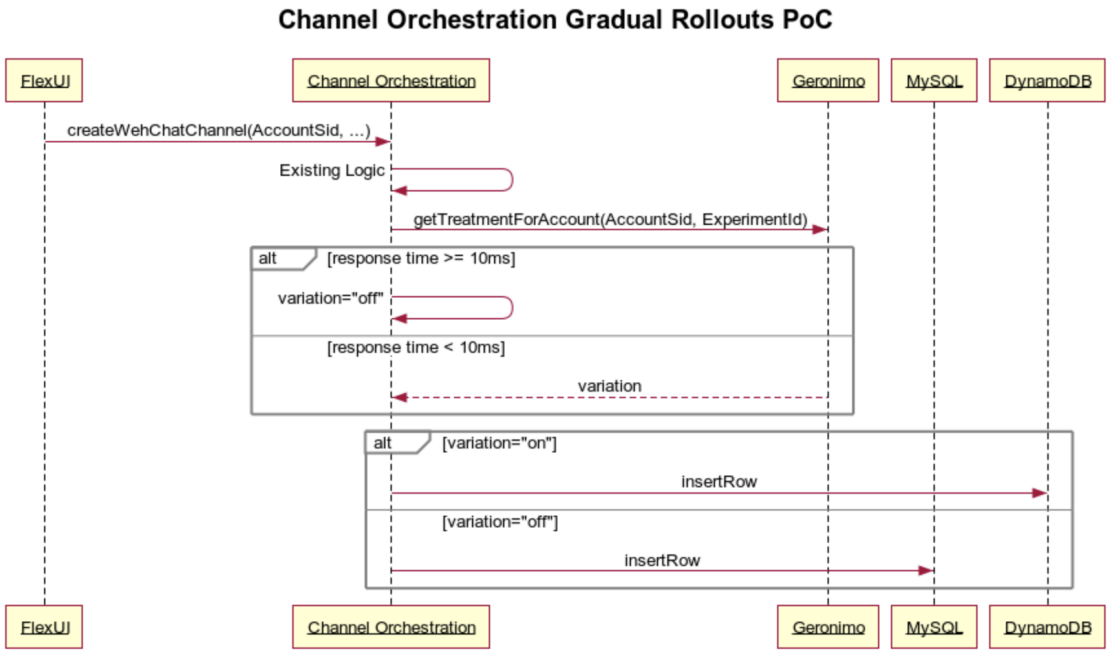

Figure 10. Channel Orchestration Gradual Rollouts PoC Sequence Diagram

Since the PoC implementation builds on the existing service, all the code changes will need to conform to the existing standards and code structure in the project. The resulting class diagram is presented in Figure 11.

The current project resides in a single code repository and the whole code is divided into autonomic modules. A new, distinct module needs to be created in the service for the Geronimo adapter layer.

`ServiceApplication` is the main class of the service where initialization of all services, resources, and repositories takes place. `GeronimoService` is an interface that will be implemented by the `GeronimoServiceImpl` class. This service will be used by a generic `ExperimentationHandler`, which includes a method `decide()`. Based on the Account SID, Experiment ID, alternative action A and an alternative action B, the `decide()` method will perform either action A or action B, depending on the variation the account belongs to.

`TempService` is a temporary class for the migration period. This class will implement the existing Service interface with temporary implementation - i.e. it will employ `ExperimentationHandler` to either call the old or the new implementation for the service.

In this specific case, there exists a `ChatUserService` interface, which includes read, write and delete logic for the `chat_users` table. This service has an existing implementation where the operations are executed in MySQL and a new implementation that includes logic for DynamoDB. For the PoC, there will be a new, temporary implementation for `ChatUserService`. This temporary implementation will use the `ExperimentationHandler` class and redirect the call to either the old MySQL or the new DynamoDB service implementation.

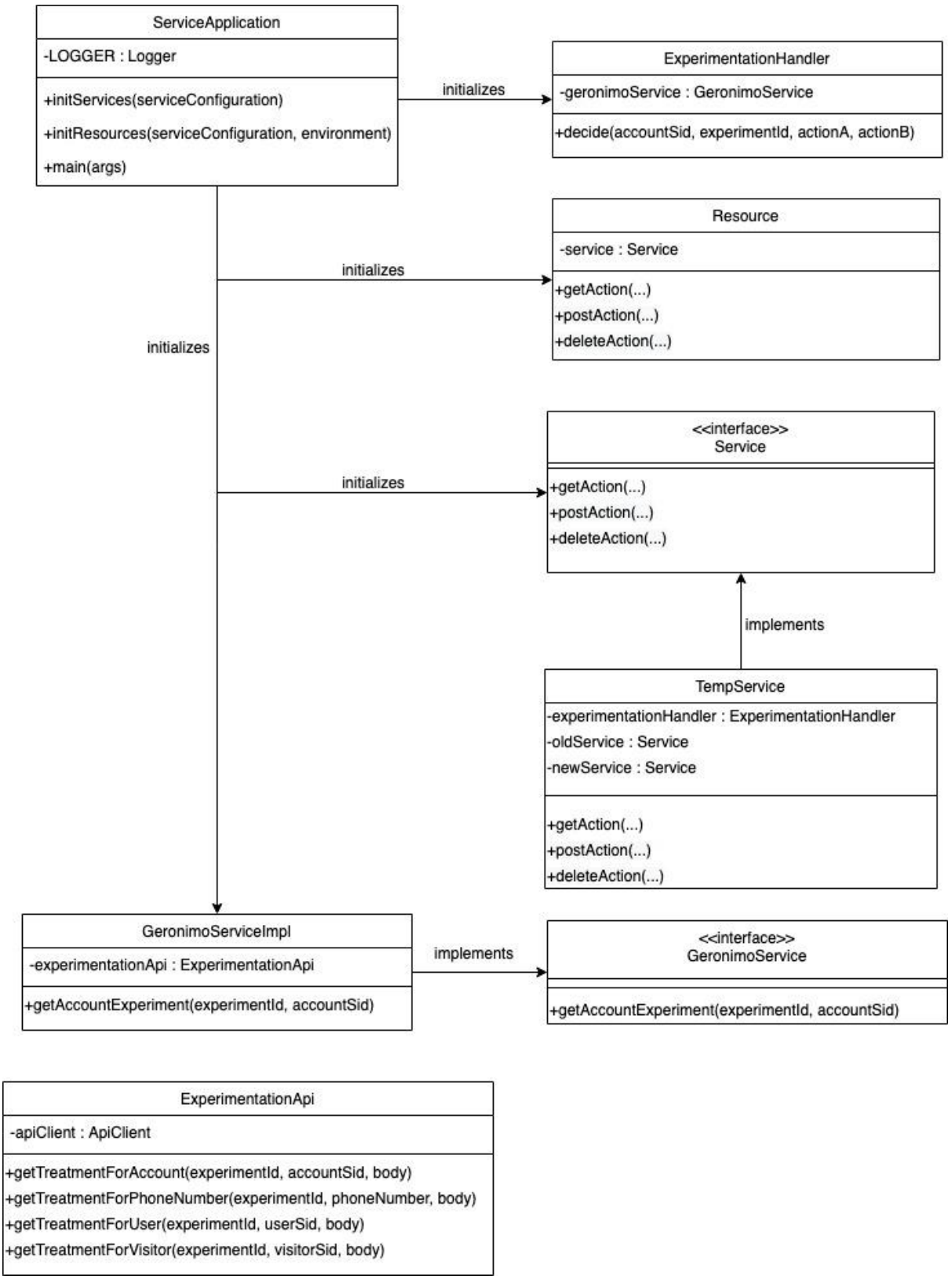

Figure 11. Class Diagram of Channel Orchestration Service Design With the PoC Implementation

4.3.2 Implementation

In order to implement the PoC, existing code of the Channel Orchestration Service was extended and modified where needed. Along with actual implementation and new class definitions, the changes include modifying service configuration, finding and adding necessary dependencies, adding and/or modifying unit, integration and load tests and their configurations.

The service is written in Java Dropwizard framework and consequently the implementation part of this thesis will also follow the same framework. Due to the fact that the service code resides in a private Twilio Flex repository, the whole code is not presented as a part of this thesis. Instead, the core segments of the implementation that are relevant to the PoC and the design described in Section 4.3.1 is presented.

Implementation of the non-blocking logic described in Figure 10 is shown in Appendix I. Using Java CompletableFuture, it was possible to implement the future which will always fail after a specified amount of time (10 milliseconds in this particular case).

The `within()` method accepts two arguments: the function that needs to be called and the time in which it needs to complete. The method `failAfter()` takes duration as an argument and schedules a `TimeoutException` to be thrown after the specified amount of time.

The method `decide()` will call the `within()` method as shown on line 7 of the Appendix I. `geronimoService` will be called to get the treatment for the account, but if for any reason it takes more than 10ms to respond, a `TimeoutException` will be thrown. The result of the `within()` method is then handled as shown starting from line 9 of Appendix I. In case the exception was thrown, variation is deemed to be "off" for this account. In case Geronimo responded successfully, the actual variation value for the account is taken (lines 14-25 of Appendix I). The specifics of the `GeronimoService` can be found in the Appendix IV.

Another important part of implementation is the generics of the method `decide()`. As seen in the method signature from lines 2-5 of Appendix I, the method is taking two callable Actions, `actionA`, and `actionB` as arguments. Interface `Action` can be seen in Appendix II.

Appendix III shows how the `decide()` method is called from a `TempService`. Lines 6 and 7 of Appendix III describe how the generic `Action` interface is implemented. Then, on line 9 of Appendix III, the `decide()` method is called and the alternative actions `a` and `b` are passed to it. In this specific case, action a means creating a chat user using the old, MySQL logic, while action b means the new, DynamoDB implementation. Although, because of the generics of the Action interface and the `decide()` method, the same classes and methods can be used for any future gradual rollout use cases.

4.3.3 Metrics

In order to have visibility over what exactly is happening in the service, relevant metrics and traceability need to be implemented. The first and the most basic step for this are the logs. Lines 11, 16, and 20 of Appendix I show how the `ExperimentationHandler` will log the information about whether the experiment was on or off for the specified account. This information will be very useful for investigations in case of service errors. Figure 17 shows an example log line produced by `ExperimentationHandler`.

For Datadog metrics, a new counter and a tag were created. Lines 12, 17, and 21 of Appendix I show how the counter is incremented with respective "on" or "off" values. The method `withTagsUnchecked()` is a part of the Twilio-internal metrics library. This library was already in use by the Channel Orchestration Service, so no new dependency was needed for publishing gradual rollouts metrics to Datadog.

After the metrics are published to Datadog and appear there, necessary graphs should be created in order to visualize these metrics. An example Datadog graph JSON is shown in the Appendix V. The graph that this JSON represents is shown in Figure 15. `experiment-id` in the Appendix V is the same as the one used in the Appendix I and is the published tag name.

4.4 Testing

This section describes how the PoC implementation was tested. Section 4.4.1 will present the Channel Orchestration baseline metrics. Section 4.4.2 describes the initial stage of testing in the DEV environment. Section 4.4.3 presents the final phase of testing in the STAGE environment.

4.4.1 Baseline Metrics

Channel Orchestration fleet consists of 18 nodes in production. Based on queries run in production, a single node has received a maximum of 17 requests per second during a one-month period. For the entire fleet, queries show a maximum of 176 requests per second. The average load on the whole fleet is 26 requests per second. On average, successful requests (with status code 2xx) take approximately 31 milliseconds, while non-successful ones (non 2xx) take approximately 47 milliseconds. Figure 10 shows average response times from Channel Orchestration grouped by response status codes.

average	status_group
0.030974275908760937	success
0.04721491280817798	error

Figure 12. Response times by status groups in Channel Orchestration

The service has also been load-tested, and results show that a single node can handle 680 requests per second.

Since Channel Orchestration is a critical dependency for Twilio Flex messaging, its outage can have a major customer impact. Contact center agents will not be able to join new chat/SMS channels or complete existing ones. The monthly uptime percentage threshold for this service is 99.95%. This is a public figure, meaning that it is written in Twilio Flex's public documentation. Internally, Flex services are expected to have five-nines or 99.999% availability.

4.4.2 DEV Environment

This section provides a summary of PoC testing in the DEV environment.

Before DEV deployment, all 239 service integration tests were run successfully. The next step was to run load tests.

As shown in the section 4.4.1, Channel Orchestration fleet consists of 18 nodes in production and each node can theoretically handle 680 requests per second. This translates to a

theoretical maximum of 12240 (680*18) requests per second on the whole fleet. If needed, an additional node could be added to the production fleet. An additional node would bring the allowed theoretical maximum on each node to 644 requests per second (12240/19). This is approximately a 5% decrease in a single node's load capacity.

In order for the PoC to pass the initial stage of testing, load test results after PoC deployment cannot differ from the baseline by more than 5%. This is because, with a 5% decrease in capacity, it would be possible to retain the theoretical maximum of requests per second by adding a single node in production.

Load test results on the new changes showed that the PoC can handle 653 requests per second per node. This is nearly a 4% decrease which is within the range of the acceptance criterion of 5%.

In reality, as shown in the Section 4.4.1, Channel Orchestration single node has received a maximum of 17 requests per second in one month period. This is only 2.5% of the available load capacity, meaning that there is a big overhead from used versus available capacity. If there was not so much overhead and there was a need to keep the same capacity, an additional node could be needed in production. This would mean an additional monthly infrastructure cost, but when compared to total infrastructure costs for running the Channel Orchestration service, this additional cost would be minimal. To summarize, load test results can be considered successful.

4.4.3 STAGE Environment

Before the changes are deployed in the STAGE environment, service cluster tests are run to verify that no breaking changes are being introduced. PoC implementation passed this step and was deployed to the STAGE environment.

In the STAGE environment, 10 test accounts were created. Four of them were artificially labeled as regular customers, three as big customers and other three as High Visibility Customers.

In the Split console two test segments were created. One segment for "HVCs", and another for the "big customers". Then, in the experiment settings, a targeting rule was created. The rule can be seen in Figure 13. The rule says to serve the treatment "on" to an account in case it is neither in HVC nor in the big customers segment.

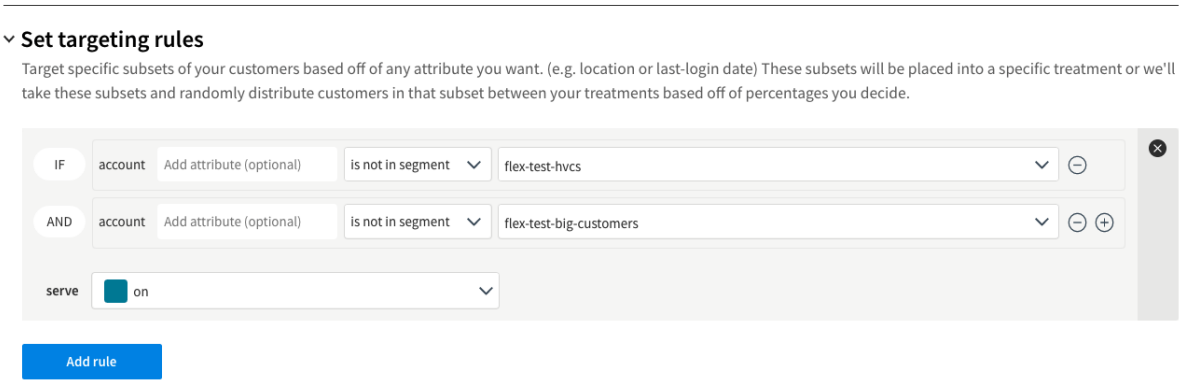

Figure 13. Experiment targeting rule

In addition, traffic allocation of 50% was set as shown in Figure 14.

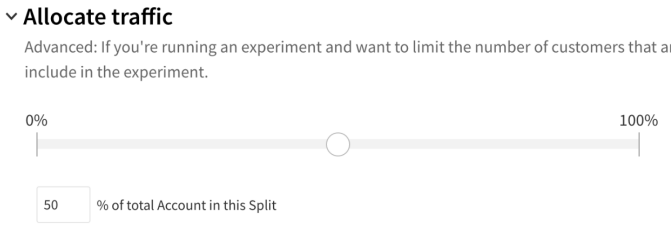

Figure 14. Traffic allocation in Split console

According to such configuration, only a random half of the total traffic will be included in the experiment. As a result, with the initial configuration shown in Figures 13 and 14, three of the "regular customers" were served a new feature and one was served an old, MySQL feature. As desired, "big customers" and "HVCs" were not exposed to the new feature at all.

Next, the traffic percentage was increased to 100%. With such configuration, all "regular customers" should get the new DynamoDB behaviour. Results show that indeed, the one "regular customer" not targeted in the first iteration was now also served a new behaviour. For all other accounts ("big customers" and "HVCs") the behaviour remained the same.

Since no errors were observed, the Split targeting rule was updated so that the new functionality would also be exposed to "big customers". The updated targeting rule is shown in Figure 15.

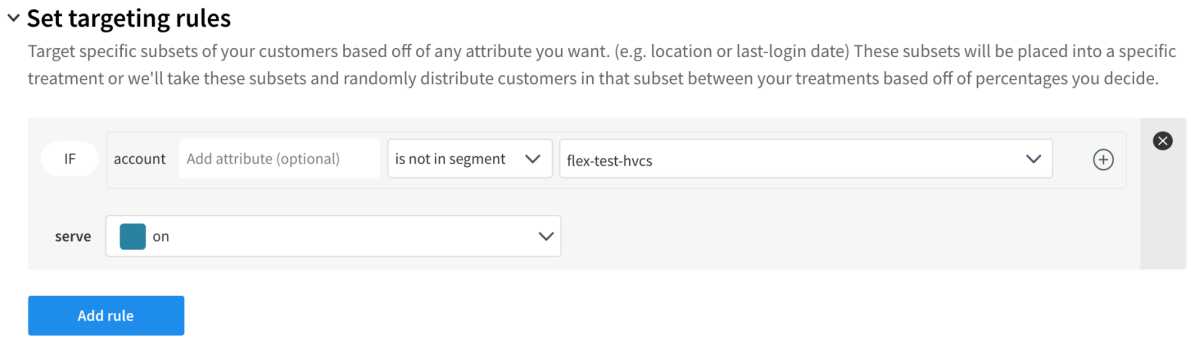

Figure 15. Targeting rule to include "big customers" in the treatment

Indeed, after the rule update "big customers" were served the new DynamoDB version of the Channel Orchestration Service. As expected, "HVCs" still remained in the old MySQL behaviour.

Finally, the targeting rule was completely removed and the default treatment and default rule were set to "on" as shown in Figure 16.

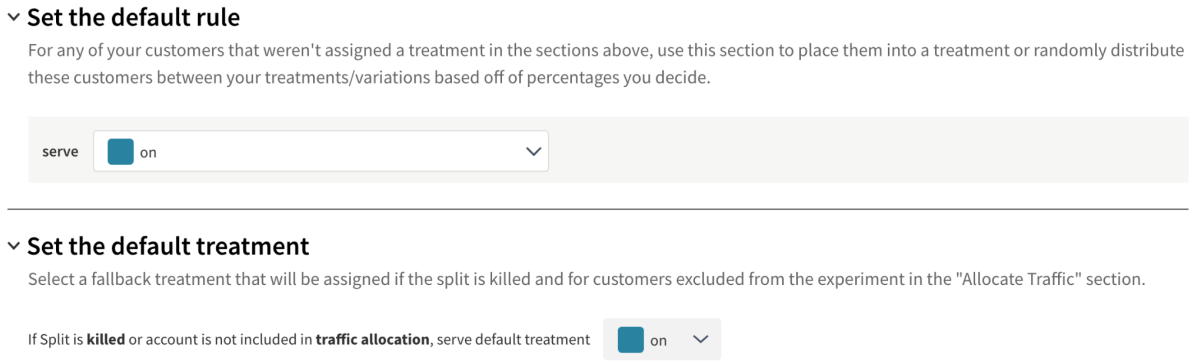

Figure 16. Default rule and treatment update

Expectation after these changes was that all customers regardless their "rank" would be served the new, DynamoDB version. Results show that indeed, after the changes all test accounts were served the new version.

Next, it is necessary to chaos test the PoC. This means that Geronimo would be artificially disconnected from the Channel Orchestration Service to verify that Geronimo outage does not cause Channel Orchestration outage, and that as an insurance measure, all the accounts will be served an old version of the service (have treatment "off"). Indeed, the assumption holds. Disconnecting Geronimo from the Channel Orchestration Service simply results in the traffic going back to the old behaviour (accounts having treatment "off").

It is also important to mention that rolling back the changes in case something goes wrong is as easy as rolling them out. Simply changing the targeting rules or the default treatment from the Split console will roll the behaviour back to the old MySQL feature.

As of the traceability of results, Figure 17 shows an example log line from the Channel Orchestration Service. As it can be seen, the message includes the information about account variation ("on" or "off"). It is also important to note that the log contains `account_sid` and `request_sid` parameters from the context. This is very useful for debugging purposes and allows querying for the logs by account or request IDs.


```
{
  "PARTITIONDATE": "2021-06-30",
  "level": "INFO",
  "message": "flex-channel-orchestration-test-split-stage is OFF for AC7342e9713fa896e4af00c654c444aada",
  "request_sid": "RQb794233ec444dadafd89c7f6b33d9420",
  "account_sid": "AC7342e9713fa896e4af00c654c444aada",
  "logger": "flex.channel.orchestration.geronimo.adapter.impl.ExperimentationHandlerImpl"
},
```


Figure 17. Example log line from the PoC

Figure 18 shows an example of the Datadog chart displaying the call counts where the variation was "on" or "off".

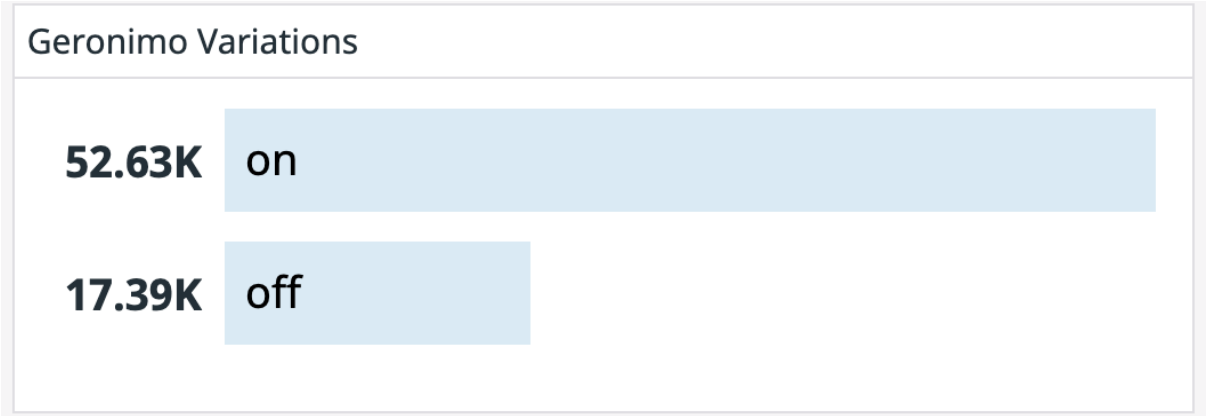

Figure 18. Geronimo variations chart in Datadog

The chart shown in Figure 15 is a simple example portraying that the Geronimo tagged metrics are available. They can be used in various ways if/when desired by the engineers.

4.5 Evaluation

Even though the PoC has not been deployed to the STAGE environment for a significant amount of time, some comparisons can still be made on the average response times. Figure 16 shows the result of a query run on the STAGE environment logs and displays average response time for 2xx and non-2xx responses before and after the PoC deployment. Average response time for successful requests was approximately 31ms and 47ms for failed requests before the integration. After the PoC deployment, successful request average time became approximately 50ms and approximately 48ms for failed requests. This is a 19ms difference for successful requests and a 1 ms difference for erroneous ones. The query for the results shown in Figure 19 can be seen in the Appendix VI. While this increase is not in violation of the Channel Orchestration SLA, it is still nearly a 60% increase and there might be a bigger difference generated over time. On the other hand, the difference might balance out and get closer to the amount before the deployment. At this point, it is hard to consider this data as a clear success or failure. There is more time and data points needed to make a judgement.

average	status_group	PoC
0.049511671144246365	success	after
0.04817833477384039	error	after
0.03097834124996278	success	before
0.047214912808177995	error	before

Figure 19. Average response times before and after the PoC deployment

Another evaluation criterion described in Section 3.6 is that the integration needs to be reliable. Taking load tests as a measure for this, it can be considered that the PoC is reliable since the load test results satisfy the requirements. In addition, all service integration tests and cluster tests are passing.

Section 4.4.3 has also described the chaos test where Geronimo was artificially disconnected from the Channel Orchestration Service. Testing showed that Geronimo outage does not cause Channel Orchestration outage, meaning that the integration can be deemed robust.

In addition, it is important to evaluate metrics' transparency. As shown in the Section 4.4.3, service logs and Datadog metrics provide visibility over general Geronimo usage statistics as well as visibility over the treatment for specific accounts and requests. Current logging and metrics can be easily improved further according to the engineers' needs.

Finally, one of the most important criterion for evaluating is whether the integration is beneficial or not. Since the integration has not actually been used in production and also it has not been deployed to STAGE for a significant amount of time, there is no objective data available that would measure PoC benefits. Because of this, PoC benefits will be evaluated subjectively. For this purpose, a senior engineer from the Twilio Flex team was interviewed. Interviewee was presented with testing and evaluation results. During the unstructured interview it was elicited that the integration matches the original needs for Twilio Flex. It has a potential to easily roll out the changes gradually, but what's more important, roll them back in no time if need be.

As of the load test results, the interview verified that the 4% decrease in capacity falls under the acceptable threshold and is expected with introducing a new dependency. In case an additional instance is needed in production to even out the capacity, this additional cost is minimal when compared to the overall infrastructure cost for the service.

For response time increase, the interviewee agreed that there is not yet enough data to say if the results are bad. Channel Orchestration SLA is 99th percentile of requests to be completed under 500ms, and PoC results fall well under that expectation.

5 Lessons Learned

This section will summarize the lessons learned while implementing the PoC. The learnings are presented as a guide. By following this guide, any team inside Twilio will be able to reproduce the PoC results in their own service.

Given that Split console access has been granted to an engineer, in order to reproduce the PoC implementation presented here, it is suggested to:

- Identify the behaviour under treatment. What are the two alternative (control/treatment) behaviours?
- Due to the generics of the implementation in the Channel Orchestration service, classes and methods already defined there can be used for any other use case as well.
    - `ExperimentationHandler` just needs two alternative behaviours as input.
    - The amount of time to wait for Geronimo's response is configurable.
    - Experiment ID is configurable.
- Add the experimentation metrics in the list of the service metrics published to Datadog
    - Actual tagging and creation of the metric is already taken care of by the generic `ExperimentationHandler`
- Add Geronimo as a dependency to the service
- Create and configure an experiment in the DEV-STAGE environment from the Split console.
- Add test treatment and control accounts and start testing.

The key takeaway from this guide is that the PoC created in the scope of this thesis is generic enough so that it can be used by different Twilio Flex backend services as well. Of course, it is not an universal solution but the current solution is easily extendable. This should significantly reduce the engineering effort needed in order to start gradual rollouts with Geronimo.

6 Conclusion and Future Work

The goal of this thesis was to implement a proof of concept of high availability deployments in Twilio Flex. This goal was achieved by first finding out that gradual rollouts using feature flags is the best deployment strategy for current scope in order to achieve high availability and de-risk critical deployments. Next, a suitable gradual rollouts tool available inside Twilio was chosen and integrated with a selected Twilio Flex service.

PoC testing has shown that high availability is indeed achievable using this approach since rollback of the changes can be done in seconds if any breaking changes are introduced. Also, there is bigger control over who gets the new feature first and last. The PoC allows engineers to roll the changes out to the High Visibility Customers last.

Objective and subjective evaluation showed that the integration is reliable, robust, fast, and has a potential to deliver big benefits to current processes in Twilio Flex.

For the future, it is expected that the metrics and logs will be updated as actual production usage will better show what really is beneficial for the engineers. Also, with usage, the generic `ExperimentationHandler` will probably be extended to suit even more use cases.

To summarize, the thesis goal was reached by developing the PoC and assessing its production readiness. Next steps and improvements will reveal themselves as the integration gets actual production usage over time.

References

1. Twilio. (n.d.). TWILIO FLEX. Retrieved April 5th, 2021, from https://www.twilio.com/flex
2. Hodgson, P. (2017, October 9). Feature Toggles (aka Feature Flags). https://martinfowler.com/articles/feature-toggles.html
3. Mahdavi-Hezaveh, R., Dremann, J. & Williams, L. Software development with feature toggles: practices used by practitioners. Empir Software Eng 26, 1 (2021).
4. Seven, D. (2014, April 17). Knightmare: A DevOps Cautionary Tale. https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/
5. Szulik, M. (2017, May 2). Colorful deployments: An introduction to blue-green, canary, and rolling deployments. https://opensource.com/article/17/5/colorful-deployments
6. C. K. Rudrabhatla, "Comparison of zero downtime based deployment techniques in public cloud infrastructure," 2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Palladam, India, 2020, pp. 1082-1086, doi: 10.1109/I-SMAC49090.2020.9243605.
7. Amazon.com, Inc. (2021). Rolling Deployments. https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/rolling-deployments.html
8. Kohavi, R. & Longbotham, R. (2017). Online Controlled Experiments and A/B Testing. 10.1007/978-1-4899-7687-1_891.
9. Tremel, E. (2017, November 21). Six Strategies for Application Deployment. https://thenewstack.io/deployment-strategies/
10. Budinski, F. (2017, June 14). Canary Deployments using Istio. https://istio.io/latest/blog/2017/0.1-canary/
11. Split. (n.d.). Frequently Asked Questions. Retrieved February 27th, 2021, from https://www.split.io/faq/
12. Unleash. (n.d.). Activation Strategies. Retrieved March 1st, 2021, from https://docs.getunleash.io/docs/user_guide/activation_strategy
13. CloudBees. (n.d.). Split Rollouts. Retrieved March 1st, 2021, from https://docs.cloudbees.com/docs/cloudbees-feature-management/latest/feature-releases/percentage-rollouts
14. Glover, A. & Probst, K. (2018, April 27th). Tips for High Availability. https://netflixtechblog.medium.com/tips-for-high-availability-be0472f2599c
15. Graff, M., Sanden, C. (2018, April 10th). Automated Canary Analysis at Netflix with Kayenta. https://netflixtechblog.com/automated-canary-analysis-at-netflix-with-kayenta-3260bc7acc69
16. Ramensky, P. (2019). Amazon's approach to high-availability deployment [Video]. YouTube. https://www.youtube.com/watch?v=bCgD2bX1LI4&ab_channel=AWSEvents

Appendix

I. Asynchronous Timeouts

```
01: @Override
02: public <T> CompletionStage<T> decide(final String experimentId,
03:                                      final AccountSid accountSid,
04:                                      final Action<T> actionA,
05:                                      final Action<T> actionB) {
06:
07:     return within(geronimoService.getAccountExperiment(experimentId,
accountSid),
08:         Duration.ofMillis(10L))
09:         .handle((experiment, throwable) -> {
10:             if(throwable != null) {
11:                 LOGGER.info(String.format("%s is OFF for %s", experimentId,
accountSid));
12:
taggedCounter.withTagsUnchecked(Collections.singletonMap(experimentId,
"off")).inc();
13:                 return actionA.action();
14:             } else {
15:                 if(experiment.getVariation().equals("on")){
16:                     LOGGER.info(String.format("%s is ON for %s",
experimentId, accountSid));
17:
taggedCounter.withTagsUnchecked(Collections.singletonMap(experimentId,
"on")).inc();
18:                     return actionB.action();
19:                 } else {
20:                     LOGGER.info(String.format("%s is OFF for %s",
experimentId, accountSid));
21:
taggedCounter.withTagsUnchecked(Collections.singletonMap(experimentId,
"off")).inc();
22:                     return actionA.action();
23:                 }
24:             }
25:         }).thenCompose(a -> a);
26: }
27:
28: private static final ScheduledExecutorService scheduler =
29:     Executors.newScheduledThreadPool(
30:         1,
31:         new ThreadFactoryBuilder()
32:             .setDaemon(true)
33:             .setNameFormat("failAfter-%d")
34:             .build());
35:
36: private static <T> CompletableFuture<T> failAfter(final Duration duration) {
37:     final CompletableFuture<T> promise = new CompletableFuture<>();
```

```
38:     scheduler.schedule(() -> {
39:         final TimeoutException ex = new TimeoutException("Timeout after " +
duration);
40:         return promise.completeExceptionally(ex);
41:     }, duration.toMillis(), MILLISECONDS);
42:     return promise;
43: }
44:
45: private static <T> CompletableFuture<T> within(final CompletableFuture<T>
future, final Duration duration) {
46:     final CompletableFuture<T> timeout = failAfter(duration);
47:     return future.applyToEither(timeout, Function.identity());
48: }
```

II. Action Interface

```
01: public interface Action<T> {
02:     CompletionStage<T> action();
03: }
```

III. TempService, Calling the decide Method

```
01: @Override
02: public CompletionStage<ChatUser> create(final ChatUser chatUser,
03:                                         final RequestContext context) {
04:     final String accountSid = context.getAccountSid();
05:
06:     final Action<ChatUser> a = () -> oldUserService.create(chatUser,
context);
07:     final Action<ChatUser> b = () -> newUserService.create(chatUser,
context);
08:
09:     return experimentationHandler.decide(EXPERIMENT_ID, new
AccountSid(accountSid), a, b);
10: }
```

IV. GeronimoService and its Implementation

```
01: public interface GeronimoService {
02:     CompletableFuture<Experiment> getAccountExperiment(final String
experimentId,
03:                                                        final AccountSid
accountSid);
04: }
```

```
01: public class GeronimoServiceImpl implements GeronimoService {
02:     private final ExperimentationApi experimentationApi;
03:
```

```
04:     public GeronimoServiceImpl(final ExperimentationApi experimentationApi)
{
05:         this.experimentationApi = checkNotNull(experimentationApi,
"experimentationApi has not been initialized");
06:     }
07:
08:     @Override
09:     public CompletableFuture<Experiment> getAccountExperiment(final String
experimentId,
10:                                                               final
AccountSid accountSid) {
11:         checkArgument(experimentId != null, "experimentId can't be null");
12:         checkArgument(accountSid != null, "accountSid can't be null");
13:         return experimentationApi.getTreatmentForAccount(experimentId,
accountSid.getValue())
14:             .thenApply(ApiClient.ApiResponse::getData)
15:             .thenApply(variation -> new
Experiment(variation.getVariation()));
16:     }
17: }
```

V. Example Datadog Graph JSON

```
01: {
02:     "viz": "toplist",
03:     "requests": [
04:         {
05:             "formulas": [
06:                 {
07:                     "formula": "query1",
08:                     "limit": {
09:                         "count": 10,
10:                         "order": "desc"
11:                     }
12:                 }
13:             ],
14:             "queries": [
15:                 {
16:                     "query":
"sum:flex.channel_orchestration_service.ExperimentationHandler.count{$env} by
{experiment-id}",
17:                     "data_source": "metrics",
18:                     "name": "query1",
19:                     "aggregator": "last"
20:                 }
21:             ],
22:             "response_format": "scalar",
23:             "conditional_formats": []
24:         }
25:     ]
```

```
26: }
```

VI. The query for Average Response Times Before and After the PoC Deployment

```
1: SELECT AVG(CAST(requestTime as FLOAT64)) as average,
2:    IF((CAST(status as INT64) >= 200 AND CAST(status as INT64) < 300),
'success', 'error') as status_group,
3:    IF((PARTITIONDATE>'2021-06-29'), 'after', 'before') as PoC
4: FROM `stage.flex_channel_orchestration_channel_orchestration_access`
5: WHERE requestTime is not null
6:    AND uri != '/healthcheck'
7: GROUP BY status_group, PoC
8: ORDER BY average
```

VII. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Sophio Japharidze**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, **High Availability Deployments at Twilio Flex - a Case Study**, supervised by **Dietmar Pfahl**.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Sophio Japharidze

05/07/2021