DC4400 Project Report

Extension of the BBC iPlayer off product schedules system to process change notifications

Oliver Matthew Bowker (220263618)

February 29, 2024

Contents

List of Figures

List of Tables

1 Executive Summary

2 Introduction

The BBC is now over 100 years old (BBC, 2022) and is well known for it's TV channels and radio stations that are broadcast over the airwaves (Pilnick, Baer, 1973, p.3) to peoples homes across the UK. However this old way of broadcasting, sending out airwaves on a certain frequency to an antenna, is becoming less popular in the modern age of the internet. A study done by Ofcom showed that people *'watched on average about 16% less broadcast TV between 2019 ... and 2022'*, with viewing *'decreasing by 47%'* (Ofcom, 2023, p.7) between ages 16-24. In addition another study carried out by media analyst firm Ampere found that in 2021 37% of people claimed to watch no linear TV, this increased to 45% by 2023 (Ampere Analysis, 2023).

This fall correlates with the significant rise in internet enabled TVs in the home, with statista finding that *'In 2014 just 11 percent of households in the UK owned a Smart TV, whereas, in 2023, nearly 74 percent of households reported owning a Smart TV.'* (Statista, 2023).

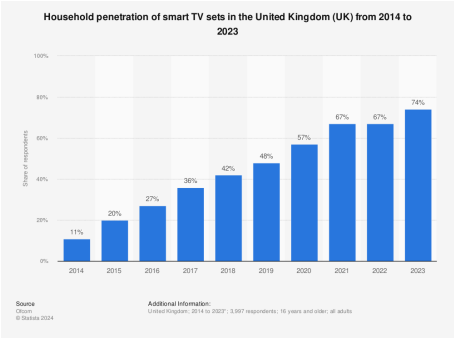


Figure 1: Bar chart showing growth of smart TVs in UK homes (Statista, 2023).

Some of these devices still support OTA broadcasts, however devices like the Amazon Fire TV stick and Googles Chromecast, are purely internet based; However they do offer a *'guide/epg'* section with Amazon having a development guide (Amazon, 2021) on how to integrate with it. Director general of the BBC, Tim Davie, in 2022 stated:

> *'The vision is simple: from today we are going to move decisively to a digital-first BBC'* (Davie, 2022)

This statement highlights the goal to put more organisational focus on these new forms of media and internet enabled devices.

This report will discuss an upgrade carried out to the BBCs *'off-product'* schedules system, responsible for delivering up to date schedules to partners such as Freeview, Amazon and more. First I will give some background on the project, where I will discuss topics including storage Solutions and how they can work in parallel/multi-threaded systems, and strategies to protect live code

systems in a CI/CD environment. I will also give some background on the starting architecture of the system and how the changes align with the BBCs and teams OKRs (Sparks, 2024).

Following that, I will discuss the work that was done. This will be broken down into 5 sections that align with our teams ways of working flow.

1. Requirements and epic creation
2. Investigation and Spike
3. Slicing and task/ticket creation
4. Development of software
5. Releasing of software

I will then talk about the outputs of the project. Theses will include burn-up charts for the projects, dashboards created, documentation of the final architecture and a description of the final product.

Finally I will discuss potential improvements for future iterations. This will range between small code changes to a complete re-architecture of the system.

3 Background

In this section I will discuss the background work and research done for this project. I will start by disusing my teams place in the organisation and our OKRs, explaining how this project helps us hit these objectives. I will then outline the current architecture and the initial design for the project. Finally I will discuss some areas of interest around project, these include cloud computing, database parallelisation strategies and CI/CD challenges.

3.1 Organisation and Team

The BBC is broken into multiple layers with different responsibilities and goals. I am in a team called *SpaceChimps* which is part of the partnerships group, which itself is in the product group.

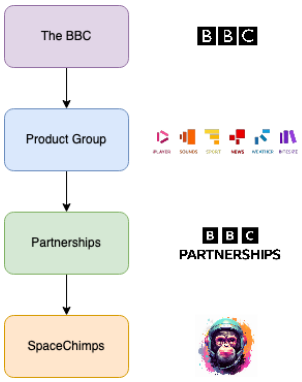


Figure 2: Image showing SpaceChimps place in the BBC (Bowker, 2023).

Our main aim as a team is to provide data to partners that they can use on their devices to promote BBC content. The project described in this report does just that by providing schedules for live content to partners. This aim fits directly into Partnerships objectives:

- The main objective of the schedules pipeline was to deprecate a service called *nitro*. This is an external platform the BBC used to distribute some of its data to partners. However this was a costly solution and meant we didn't have full control over the process. This projects aim is to improve this system allowing us to have better metrics and control over the data we share, providing the most up to date and relevant schedules.
- It helps drive growth as we are able to get content out to more people on more devices, increasing exposure to the BBC.
- It helps us improve our partner experience by working with them on integrating the data into their feeds.

- This project reduces the total time processing data, which therefore reduces our costs and makes us more sustainable.

All objectives can can be seen in **Appendix A** (BBC Partnerships, 2023).

As well as schedules we also provide a *'catalogue'* of episodes, series and brands that are currently available on iPlayer.

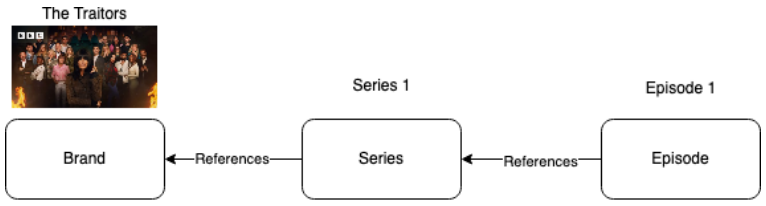


Figure 3: Image showing catalogue hierarchy and how they reference each other.

We provide both a 0 day and 8 day catalogue, these containing programme data that are available now or up to 8 days in the future. We also have an *'unfiltered'* catalogue that is not available to partners which contains all programme data with no availability limits. This unfiltered catalogue is what is used by the schedule pipeline to get it's data about episode/series/brands within the schedule.

3.2 Original Architecture

The original solution was composed of AWS services that created a pipe and filter architecture which transformed inputs into multiple outputs (Somerville, 2016, pp.182-183) that can be used by partners. The figure below shows this architecture.

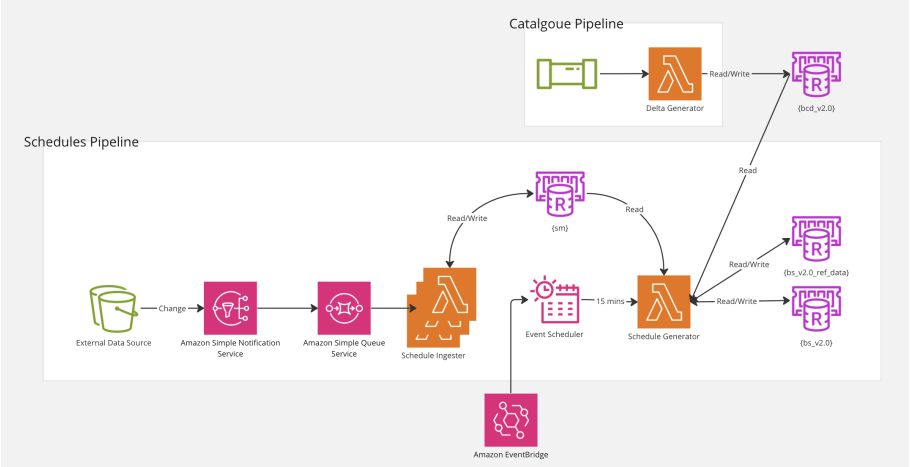


Figure 4: Image showing the initial architecture.

A pipe and filter architecture is achieved with the combination of AWS lambda (Amazon Web Services, 2024a) with updates being triggered by a publish/subscribe model (Somerville, 2021, p.179) achieved through AWS' Simple Notification Service (SNS) (Amazon Web Services, 2024b) and Simple Queue Service (SQS) (Amazon Web Services, 2024c) with the former publishing and the latter subscribing. The lambda only runs when a message is published to the SQS, this triggers the lambda and the pipeline begins precessing the new message.

The data is updated by an external system in AWS S3 (Amazon Web Services, 2024d), this publishes a message that the data has changed, our first lambda (the ingester) receives this message and processes/stores it into a *'common'* model that's used internally only. Our internal model is then up to date, however our partner facing model, created by the schedule generator, is not. In the original system this lambda was not driven by events, but instead ran every 15 minutes and processed all the schedules in one go, whether they had updated or not.

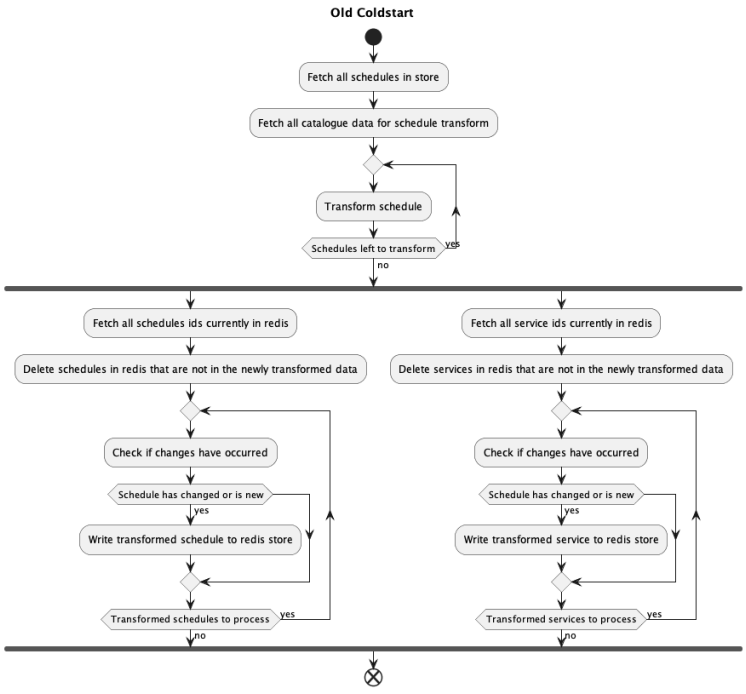


Figure 5: Activity diagram showing schedule generators logic.

This could leave partners with data up to 15 minutes out of date, dependant on when they retrieve the data, resulting in end users being shown incorrect schedules.

3.2.1 Initial design solution

The schedules pipeline also requires data from our catalogue pipeline, for titling, descriptions, viewer discretion warnings and subtitles. For this reason it needs to be alerted when catalogue data changes as well as when schedules change. An initial design had already been complete before the work started by another member of the team.

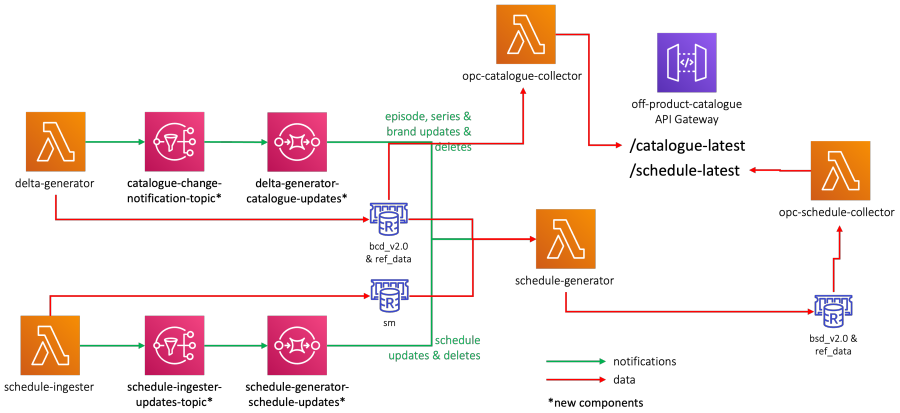


Figure 6: Image showing the initial updated schedules design (Lloyd, 2023).

The above diagram has been edited down to include only work related to the project, a full diagram including notifications being directly sent to partners can be found in **Appendix B**. This diagram also includes our endpoints and collectors that partners use to retrieve the data. This proposed system uses the pub/sub model described earlier and will subscribe to both catalogue and schedule events.

Alongside the architectural design an algorithm was proposed for these events.

- **For schedule updates** - We want to update the partner facing model of the schedule linked in the notification. This schedule has a list of broadcasts that map to episodes in the catalogue. A list should be maintained within each episode to create a link between both, allowing episode updates to trigger updates to schedules that reference them. All catalogue data referenced in a schedule should be copied over to the schedules keyspace, leaving the catalogue keyspace and items untouched.
- **For schedule deletes** - Remove schedule from store, and remove schedule reference from list contained within each episode that schedules references in its list of broadcasts.
- **For catalogue/programme updates:**

  For episodes - Update episode in store, update each schedule that is linked in episodes broadcast list.

For series/brands - Get all episodes linked to either the series or brand, update each schedule that is linked in each episodes broadcast list.

Flow diagrams can be found in **Appendix C** (Lloyd, 2023).

With the addition of schedules and broadcasts lists in episodes, our relations between types objects can now be described in the below ERD diagram.

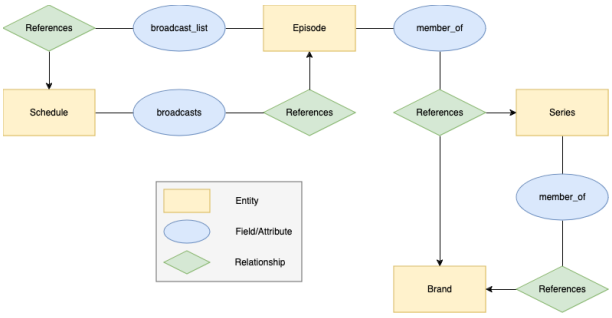


Figure 7: Relationships between types of data offered to partners.

Episodes and schedules will now link to each other via their associated fields.

4 Research

I will now explore research done relating to the project specified. This will outline some design decisions that were made and link to other sections of the report where they may be explored more. This section will explore parallelism and thread safety focusing on data stores as well as CI/CD, it's pros and cons when deploying to live systems and how we as a team worked around it.

4.1 Storage Solutions and Parallelism

Currently the schedule pipeline consists of two components, the ingester, followed by the schedule generator. The first part of this pipeline is parallelised, multiple lambdas can be ran at the same time to insert data into the redis. This is a harder task for the schedule generator to do, as it needs update a list of linked schedules to an episode in the redis store. This data can be edited through multiple streams, both the schedule catalogue pipelines, meaning the array could easily become incorrect/polluted. Sharing memory in a threaded/parallelised system is a well known challenge and you need to know when it's safe to update/edit this memories value (https://homes.cs.washington.edu/ djg/teachingMaterials/spac/sophomoricParallelismAndConcurrency.pdf). This is also known as being thread safe which can be described as *'different threads can access the same resources without exposing erroneous behavior or producing unpredictable results'* (Ugarte, 2024).

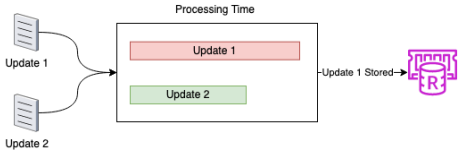


Figure 8: Diagram showing how a broadcast list of an episode can become incorrect/polluted.

4.1.1 Redis and Elasticache

We use amazons Elasticache (Amazon Web Services, 2024f) for our redis (REmote DIctionary Server) solution as it stores data in memory, which makes it really quick to retrieve stored data (IBM, 2024). This is vital for us as we store large documents that need to be retrieved and sent to partners on an API request. Redis is single-threaded but supports concurrency, *'when at least two threads are making progress'* (Oracle Corporation, 2010) which is not the same as parallelism, *'when at least two threads are executing simultaneously'* (Oracle Corporation, 2010).

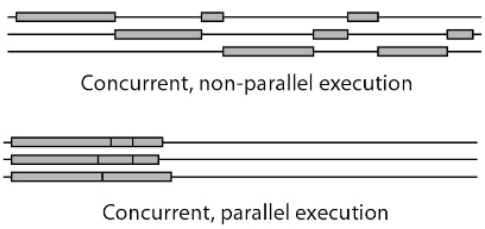


Figure 9: Difference between concurrency and parallelism.
https://books.google.ie/books/about/Introduction_to_Concurrency_in_Programmi.html?id=J5-ckoCgc3IC&redir_esc=y

This concurrency allows redis to support multiple requests at once, but it cannot do multiple operations at once; although more recent versions are allowing some safely threaded operations such as deleting records (Redis Ltd, 2024a). It also supports batch uploading and blocking commands, however these also don't help keep the data thread-safe.

- **Pipelining** - Pipelining sends a block of commands at once, however does not guarantee that commands sent are done in sequence (Redis Ltd, 2024b).

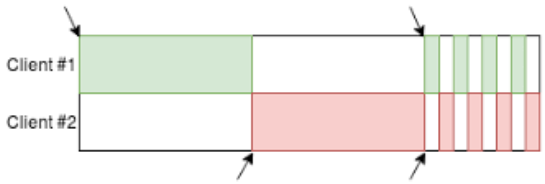


Figure 10: How redis pipelines don't guarantee sequential execution (Eyng, 2019).

- **Transactions** - Transactions are very similar to pipelines, however guarantee that the transactions commands are not interrupted by another clients requests and are therefore executed in sequence (Redis Ltd, 2024c).
- **Blocking Actions** - Blocking actions stop the current client from executing commands until the blocking action is complete. However other clients can still send requests to the server whilst this client is blocked (Redis Ltd, 2024d).

The options above still allow the previous race condition to occur, as instances of the schedule generator may vary in processing time and therefore the time of writing to the redis cannot be guaranteed to be in order of the events.

4.1.2 How thread-safety can be achieved

When researching and spiking (Visual Paradigm, 2024) the project, other technologies were found that could help offer thread safety whilst parallelising the schedule generator. These were types of store/database locking mechanisms.

- **Pessimistic Locking** - This method *'assumes that access to shared memory will be contended'* (Weston, 2011) and acquires a lock on the data to be edited. Any other client/connection attempting to edit this data must wait until this lock is released to update the data (Thornton, 2001). This can lead to issues such as deadlock which is when two clients are both awaiting on another clients lock to be released. This can end up with both clients being stuck in a endless cycle of waiting for each other (Thornton, 2001; Apache Software Foundation, 2013).

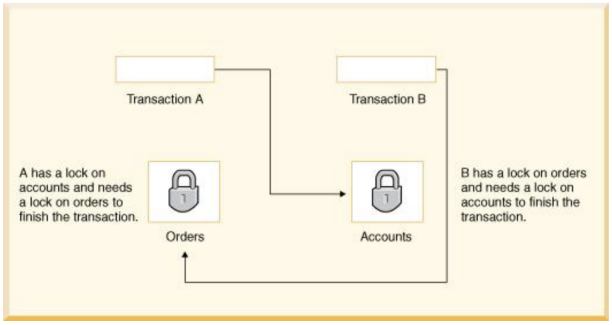


Figure 11: Example of a deadlock scenario (Apache Software Foundation, 2013).

  There is also a situation where a client locks a piece of data and for one reason or another halts, this would cause needless delay, when a timeout is specified, if a timeout is not specified then this object would be locked from changes indefinitely (Thornton, 2001).

- **Optimistic Locking** - This method *'relies on end-of-transaction validation'* (Graefe, 2016) and takes the outlook of presuming it's safe to write until the very end (Kanungo, Morena, 2023). Unlike its counterpart (pessimistic locking) it does not lock the record that is being updated. Instead before the new data is written, the original data is checked against the current data stored (Thornton, 2001). If this data doesn't match then a change has occurred during processing and the new data to be written must be re-calculated with the new changes.

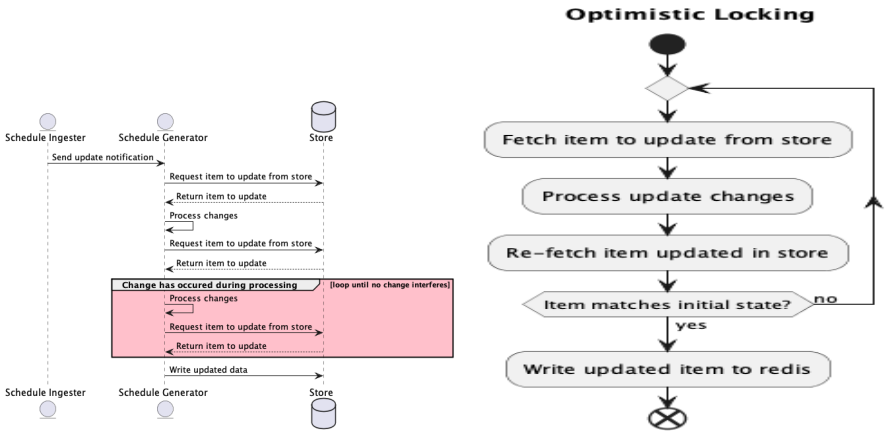


Figure 12: Sequence and activity diagrams outlining logic of optimistic locking.

> This above logic could be implemented into our redis solution. It would require either a total comparison of the object or a simple *version* field that specifies when the object has changed.

During this investigation DynamoDB (Amazon Web Services, 2024e) was highlighted as a potential option due to it supporting optimistic locking. The final decision was to not use it however and stick with the current solution as there was a want to get the project started to keep up with our roadmap and not further investigate this option. There was also more unknowns with this technology as we had never used it before. I will discuss a solution using DynamoDB in the **Future Work** section of this report.

4.2 Agile and CI/CD

As a team we follow an agile approach to software development, more accurately we follow a kanban approach. The kanban methodology *'focuses more on monitoring and improving workflows'* (Heil, C, 2022) and doesn't use concepts such as sprints from scrum (Rehkopf, 2024). Instead software created using the kanban approach is deployed/released when it's done (Rehkopf, 2024) and uses a kanban board where tickets move across columns depicting where they are currently at in development cycle (Mauvius Group Inc, 2021).

Figure 13: SpaceChimps kanban board.

This workflow fit's in well with Continuous Integrations and Continuous Deployment (CI/CD). As a team we release/build all our own code to multiple environments using Jenkins declarative pipelines (Jenkins, 2024).

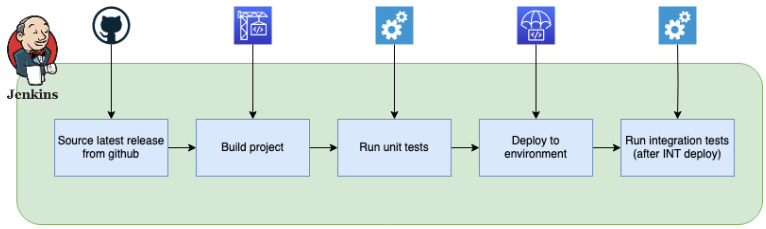


Figure 14: General pipeline flow.

Having these well defined deployment pipelines makes deploying new code easy and helps manage deployments across multiple environments as configuration can be built in to the pipeline (Rodriguez, et al, 2016). This approach allows bugs to be caught faster than when using a *'big-bang'* approach, as the bug is likely to be with a single new change (Department of Defence, 2021). The additional testing also gives more confidence in the code deployed and rollbacks are made extremely simple due to version control management.

There are of course some downsides to CI/CD, whenever an external platform, Jenkins in our case, is being used this opens up a new attack vector (NSA, 2023). OWASP keeps a list of some of these threats (OWASP Foundation, 2023), however they mostly comprise of poor authentication to CI/CD systems and attacks by people who already have access to source code and the CI/CD platform itself (employees).

Other issues include maintenance of the pipelines code/infrastructure and potential complexity (Wikström, 2019), this can be especially true when pipelines call other processes.

Code quality can become an issue, with technical debt increasing due to the encouraged continuous deployment of new software (Rodriguez, et al, 2016) and more bugs also appear to occur. One study showed that the number of bugs actually increased when using CI/CD (Fairbanks, Tharigonda, Eisty, 2023).

	CI/CD	No CI/CD
Github average issues	135.38	57.33
Gitlab average issues	52.04	17.68

Table 1: Table from study showing difference in issues found between approaches (Fairbanks, Tharigonda, Eisty, 2023).

However this same study also showed a commit velocity increase of 141%, so it could be argued that these bugs are quickly remedied due to this quicker release time. In addition to this as a team we do pull requests, another developer

checks new code before release, which should also mitigate some of the outlined issues above.

4.2.1 Software Changeover Strategies

The system that is being upgraded is in LIVE use by partners, but the work itself will take time to implement. We don't want to make changes to how the LIVE system works for partners but we do want to test that the new software works on the LIVE environment without a *big-bang* deployment.

There are three main types of changeover strategy, direct (big-bang), parallel running or a phased strategy and it's multiple variants (Banerjee, 2017).

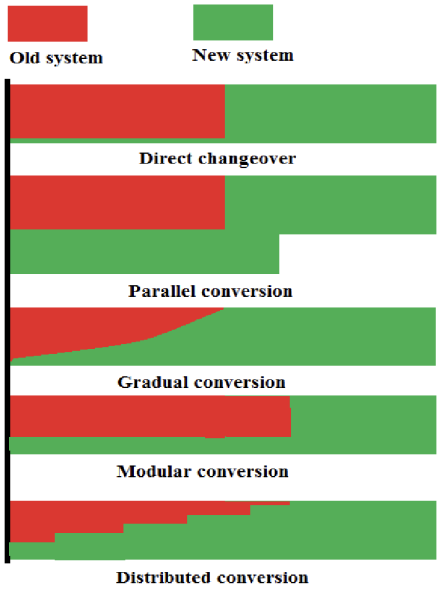


Figure 15: Timeline of different changeover strategies (Banerjee, 2017).

As discussed a direct approach is risky and could easily run into errors, a phased approach in this situation is also not valuable as the we want to remain consistent in the data we provide to all our partners, so a parallel system is our best option. Temporarily the old and new system will run side-by-side, this will allow us to write comparison tests between the old output and the new (Smyth, 2020).

In addition to having both systems the data on redis also needs to be separate as they may differ during development. This will be achieved by using different redis keyspace prefixes (IoRedis, 2024) to keep the data separate and allow a test to check both for differences (Rustagi, 2023). The partners would continue to get data from the old keyspace, when we are happy with our comparison tests the partners can be swapped over to the new keyspace and the old data can be

deleted.

5 Work Done

In this next section I will discuss the work that was carried on the project. In our team we have *'ways of working'* flowchart that helps guides us in how the project should be done throughout the team (GoRetro, 2023). Our ways of working is broken down into 5 sections, all of which will be discussed. For a full diagram of our workflow see **Appendix D**.

5.1 Requirements and epic creation

The First stage is initiation, what's the project going to be about, how it fits into the companies strategy and OKRs, requirement gathering and epic creation, an epic being a set of user stories/features (Karolis, Saulius, 2023).

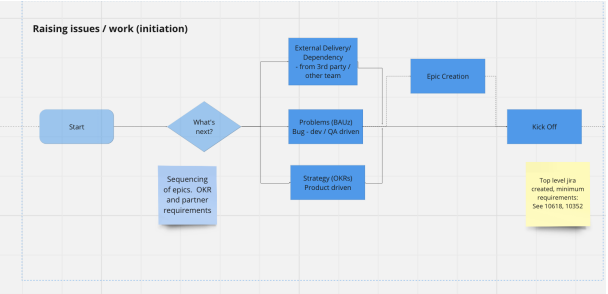


Figure 16: Initiation stage of our ways of working.

Due to complications in project allocation, I was not involved in the initial phases of development for this project. However discussions with people that were, highlighted that due to the lack of external partner interaction their wasn't much to discuss and no solid requirements were outlined, only the goal which is to allow the schedule generator to be moved from a static refresh of schedules to an event driven process.

The benefits of the project and how it achieves our OKRs has been discussed previously in the report. However I want to create some tangible requirements here using the MoSCoW framework which consists of must, should, could and won't haves (Eduardo, 2022). I will focus on software requirements and denote whether they are functional (F) or non-functional (NF) (Bigelow 2020).

1. (F) The software **Must** be able to process schedule updates.
2. (F) The software **Must** be able to process episode updates.
3. (F) The software **Must** be able to process ancestor updates.
4. (NF) The software **Must** keep stored data more relevant than the static version.
5. (F) The software **Must** still support a coldstart option.

6. (F) The software **Must** contain alarms to alert us to errors.
7. (F) The software **Should** gather metrics on processing time and types of updates.
8. (F) The software **Should** clean up unneeded data from it's store.
9. (NF) The software **Could** support parallelised processing.

In this case requirement gathering was easy as there were no partners to discuss the changes with as it's all internal and the output to partners stays the same. If this wasn't the case then requirements would have to be gathered from all partners affected or interested in the projects outcome. These requirements would be gathered through meetings and conversations with these partners individually and compromises may have to be made by both sides. This is usually the case when there is a requirement conflict, which can be described as:

> *'Requirements conflict is defined as unexpected or contradictory interaction between requirements that has a negative effect on the results (Cameron, Velthuijsen, cited in Kim et al, 2007)'*

These conflicts need to be managed well, as Kim et al (2007) warns these conflicts can *'lead to negative or undesired operation of the system'*. However certain stakeholders will often carry more leverage than another, taking into consideration the Pareto Principle (Sanders, 1987), the 80-20 rule, it's easy for an organisation to leave out requirements of smaller stakeholders. A balance must be struck here where all parties are kept happy, without damaging the original idea of the project when large stakeholders try and morph it into what they deem it should be.

In this case no conflicts occurred due to the how internalised the project is. An epic was created in Jira and theses requirements could then be explored and implemented in the next phase.

5.2 Investigation and spike

After initiation the project is investigated and then *spiked*. In this case investigation/research for the most part has been done, as we knew the system was feasible due to following the same design as our catalogue pipeline. However it was determined that a spike of the proposed algorithm would be done.

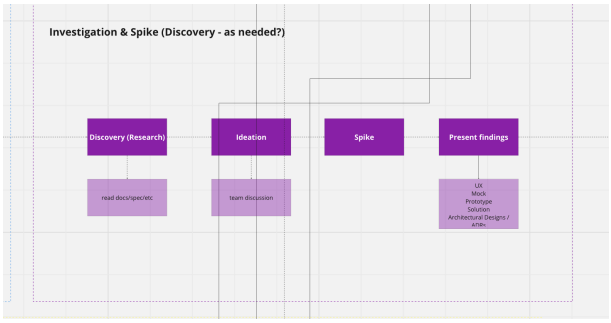


Figure 17: Investigation stage of our ways of working.

A spike is often used when there are unknowns about a project (Visual Paradigm, 2024). A study done by Hashimi and Gravell (2020) found that a majority of people in industry saw spikes as an effective tool and helped in managing risks. In another study they also stated that one of key purposes of a spike was to help guide story/ticket estimations (Hashimi, Gravell, 2019).

In the first paper it was also hypothesised that technical debt can also be lowered through the use of spikes, with technical debt be defined as:

> *'the idea that developers sometimes accept compromises in a system in one dimension (e.g., modularity) to meet an urgent demand in some other dimension (e.g., a deadline)' (Kruchten et al, 2012)*

Spikes were used by Glas and Hedén (2021) to help bring down technical debt in their study. However when thought about simply, a spike could be seen as a tool to spot the technical debt before it touches live systems. Due to a prototype being made flaws can be spotted early and ironed out before the *'real'* development work begins.

For the project a spike was carried out to test the algorithms outlined in **Appendix C** to test the throughput of the system and see if it could handle the incoming requests. We used the technical spike template recommended by Microsoft (2021), to document findings and outline the spikes objectives. Hashimi, Abduldaem and Gravell (2022) found that *'timeboxing, objectives, documentation, and clear communication'* were the key factors that lead to a successful spike.

There was no time limit set, which did lead to scope creep (Martins, 2023) and more time being spent than was necessary. It's important to remember that a spike is not meant to be the final product, however scope creep lead to

the spike being almost a fully working system, albeit unrefined and with issues. This is something as a team we have now changed in our process and will always timebox spikes in the future to prevent this.

Despite this the other key factors were well adhered to. Objectives and documentation of the spike were put into a spike document and we had communication with a more senior member of the team at all times as well as daily stand ups to communicate issues/progress on the spike. The full spike document can be seen in **Appendix E**, below are the key findings.

1. A garbage collector should be used to clean up data that is no longer referenced by a schedule. This will help lower the amount of redis sets/gets and has no negative affect on partners.

2. When a catalogue item (episode/series/brand) updates it will also need to update it's associated schedules, for titling and descriptions. This in itself is not a problem, however if the number of schedules referenced is large it can significantly slow the system down. Parallelisation should be looked into, at least at the schedule level to help ease this. Full parallelisation would require a new design to support parallelised editing of the broadcast list held in episodes (this discussed in the **Research** section).

3. Redis duplication is complex and is only needed so that episodes can reference a list of schedules that they are in. Might be worth using DynamoDB here as this will also help with parallelisation (This is explored in the **Future Work** section).

4. Additional filtering from the catalogue pipeline could be added to only send updates for items that are referenced by a schedule. This would stop lambda being ran that essentially do nothing, this is also discussed in **Future Work**.

It was determined by the team that, for now, the system should be single-threaded and stick to the original design due to time constraints. Parallelisation could be looked into in the future when we had some time. However it was agreed for a garbage collector to be used and when a batch of schedule updates is is triggered by a catalogue update that some form of concurrency would be required. This can be done with schedule objects due to the single threaded nature of the current system. As the schedule updates are being done because of a catalogue update the only fields that can change are the titles and descriptions. These do not affect or reference anything other than the object itself.

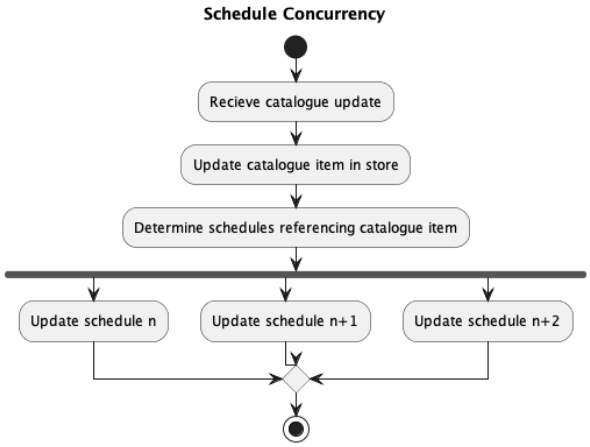


Figure 18: Simple activity diagram showing basic concurrency logic for updating schedules triggered by a catalogue update.

5.3 Slicing and kick-off

Another small section is the *'kick-off'*, usually a single meeting where the team gets together and discusses the tickets that makeup the project as a whole.

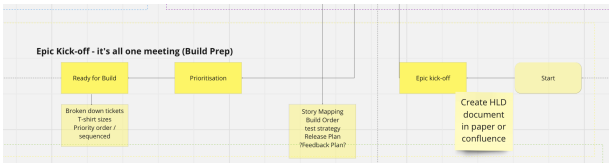


Figure 19: Kick-off stage of our ways of working.

Ideally this work has already been broken down into tasks. This is where the spike really helps, instead of guessing we are able to better understand the work that needs doing and create tasks accordingly (Hashimi, Abduldaem, Gravell, 2022). The following diagram shows the initial breakdown of tasks/tickets that were created for the project.

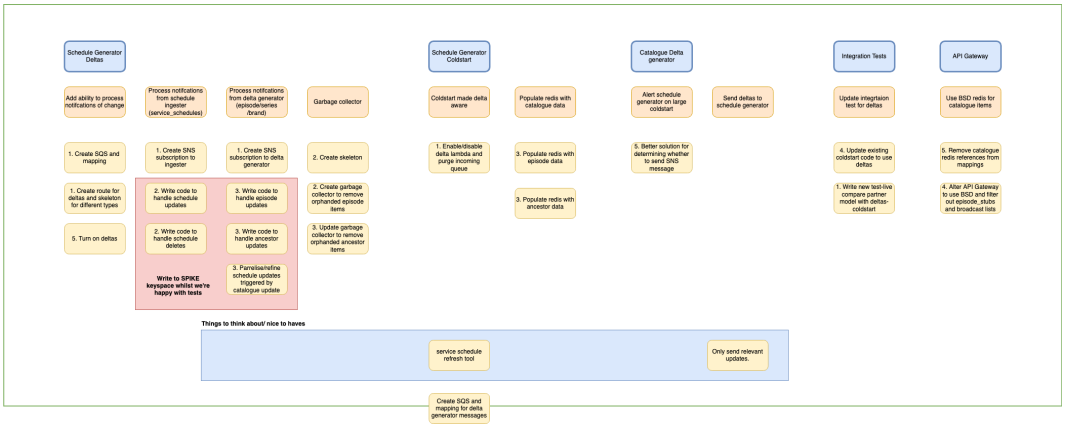


Figure 20: Pre-work slicing of work, organised vertically into components.

A work breakdown structure (WBS) is a tool used to *'break down deliverables into sub-deliverables to visualize projects'* (Raeburn, 2024) and is typically broken down into 3 levels, parents, dependencies and sub-tasks, however it can consist of as any as a team wants. For the above I kept it at 3 levels, going vertically, blue represents self-contained features/components, orange is a parent task to complete the feature and yellow is a sub-task of the parent task. Numbers on the sub-tasks represent the initial prioritisation/ordering of the tasks into slices or feature sets.

It's important to note that this is not the final list of tasks. Unknowns will always appear whilst developing the software and result in additional tasks being created. However the WBS helps refine the scope to what is needed (Burghate,

2018). As can be seen in the above figure there are 3 tasks that have been moved to a *'nice to have'* section, which have been deemed unnecessary for the projects initial completion.

Another way to visualise these feature sets is using an Archimate implementation and migration diagram. This allows the modelling of work packages, gaps, deliverables, plateaus (stages) and events (The open group, 2016). The diagram on the next page shows the project mapped out using these elements, documenting the deliverables for each feature set.

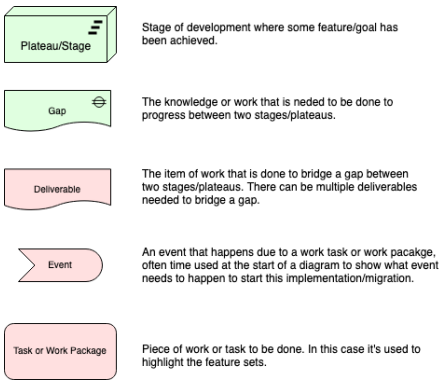


Figure 21: Implementation and migration diagram components (The open group, 2016 and Jonkers et al, 2011).

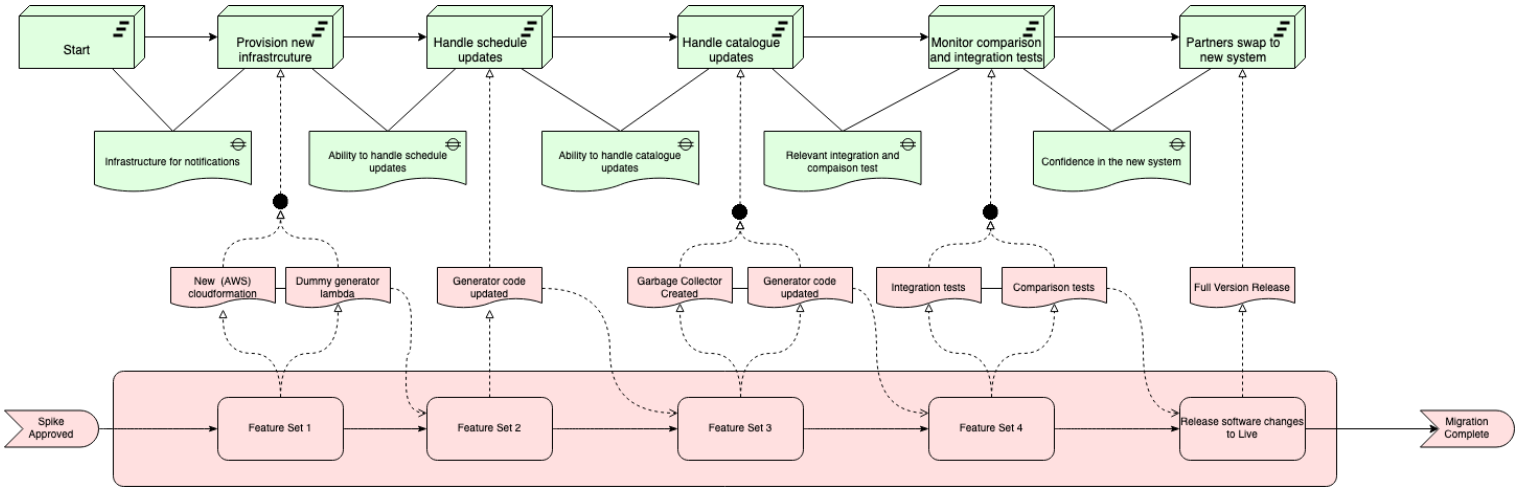


Figure 22: Figure showing Archimate implementation/migration diagram for project.

5.4 Build software

Before discussing the build stage, it's important to understand the multiple environments used in development.

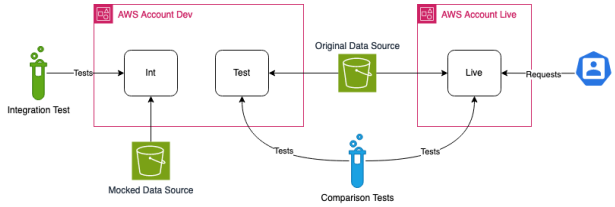


Figure 23: Diagram showing the different environments used by SpaceChimp.

The above figure illustrates where the pipelines get their data from dependant on the environment, as well as what tests access said environment. We have 3 environments, int, test and live, with test mimicking live (Wiggins, 2017). This allows us to use the test environment to protect live from any bugs or errors introduced by a task whilst also allowing the testing of outputs between old and software (Zheng, 2021). However when running on test and live we don't have control over the data source and the events it sends out to the pipeline, thus making it hard to test certain scenarios and features. This is where the int environment is used, we mock the data source used on test/live but have full control over what is added and removed, This allows us to routinely check all edge cases and features are working as expected.

The build stage consists of writing and testing the software and resembles the flow of our kanban board.

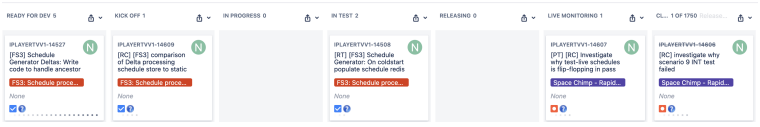


Figure 24: SpaceChimps kanban board.

- **Ready for dev** - Tasks that are ready to be picked up for development.
- **Kick-off** - Tasks that need a test kick-off, which is not the same as the previous sections kick-off. This is where developers discuss the task with a member of the test team and determine the Acceptance Criteria (ACs) and test approach for the task.
- **In Progress** - Tasks that are being developed and worked on.
- **In Test** - Task has been completed and changes are on the test environment. A member of the test team can now test the ticket based on the

previous discussions had in the kick-off. If things have changed during development, an additional hand-over with the test team is done to discuss the new changes.

- **Releasing** - After a ticket task has been tested and has met the specified ACs, the task is moved to the releasing column signifying that is ready to be deployed to the live environment.
- **Live monitoring** - Tasks in this column need to be monitored on the live environment to make sure that the change is functioning as expected.

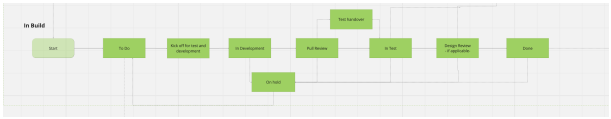


Figure 25: Build stage of our of working.

I will now discuss the 4 key components/features that were built during this stage, along with decisions and challenges that occurred during development.

5.4.1 Delta/change lambda

Before notification events were being processed, schedules relied on the cold-starting/refreshing of data every 15 minutes. The delta lambda is the the component that moves away from this paradigm and is the main component in this project. I have separated this section into the different types of notifications the component can receive, schedule updates and deletes and catalogue updates and deletes. But before delving into that, a bit of extra information is required about the build.

In the old system there was no link between episodes and schedules. We now need one, as an certain changes to episodes can change what a schedule contains. For this reason it was decided that a new field could be added to episode, the broadcast_list field. This would contain a list of all schedules that an episode was referenced in.

In addition to this, an architectural decision was made, before me taking on the project for this report, to have all schedule documents be in one keyspace, this can be seen in **Appendix F**. This making it easy to return all the relevant data on partner requests from our API and now with an additional field, making it so catalogue schemas would not have to be changed. To do this all data relating to any current schedules offered must be copied over from the catalogue redis keyspace to the schedule redis keyspace, with the broadcast_list field being added only to the schedules version of the object. The below figure shows how this works on a schedule update.

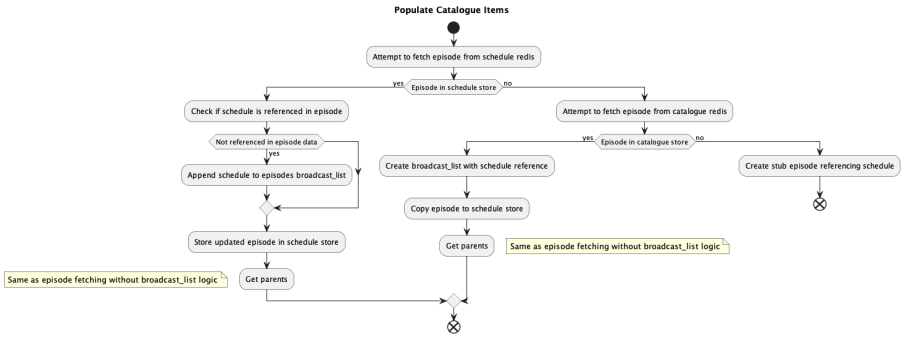


Figure 26: Activity diagram showing logic of populating schedule redis.

There's 3 options that can happen here:

1. If the episode is already in the schedules store, then we need to add a link to the current schedule being updated if it doesn't already reference it.

2. If it's not in the schedules store, but is in the catalogue store, then it needs to be copied over and have the current schedule being updated referenced in it's broadcast list.

3. If it's not in either store, then a stub must be made, this stub has a unique identifier (pid), a type of *episode_stub* and a broadcast list, that in this case holds the schedule currently being updated.

The episode stub is required, despite it containing no useful information to the schedules tilting and other fields and is also discarded when a partner requests said schedule. It is required because of the event driven nature of the new system and the broadcast list it contains. With a stub in the store, when an episode update is processed for that stub it can now also update all the schedules associated with it, immediately populating tilting and descriptions for all.

As will be discussed in the following sections, different types of notifications can affect different kinds of data. Below is a table showing the relations between types of updates and what can be affected. The garbage collector will be discussed later in the report, but is also included in the table for completeness.

	Schedule	Episode	Ancestor
Schedule Update	✓	✓ ✗	✓
Schedule Delete	✗	✗	
Episode Update	✓	✓	✓
Ancestor Update	✓		✓
Garbage Collector			✗

Table 2: Table showing how different types of notification can affect stored data (✓ updates, ✗ deletes).

5.4.1.1 Schedule Updates

Schedule updates refer to either the updating of a schedule, or the creation of a new schedule. There are some guards around whether the schedule exists, then all catalogue data associated with the schedule is copied from the catalogue store to the schedule store, if it's is not already there. This data can then be used to transform everything into the final schedule, where if any valid changes have been made, the new schedule is stored.

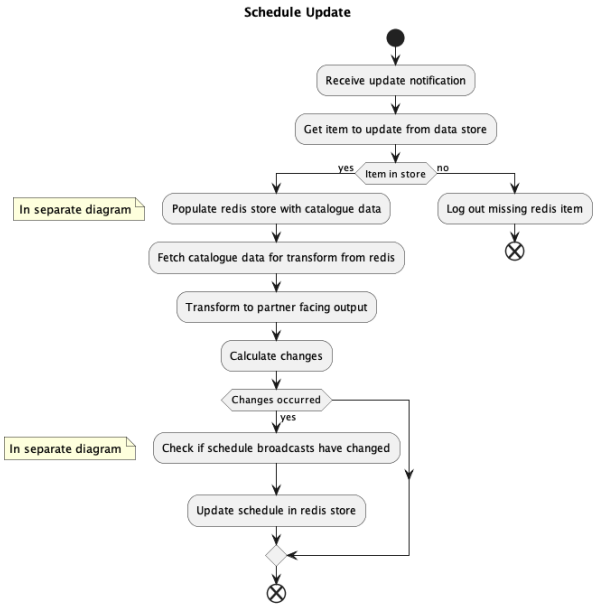


Figure 27: Activity diagram showing logic when schedule update notification received.

There is an edge case specified in the above diagram, where a broadcast within a schedule changes the episode that it's going to play. In this case we have to either tidy up the old episodes broadcast list to no longer contain a reference to the schedule, or, if that episode no longer references any schedules, remove it from the schedule store.

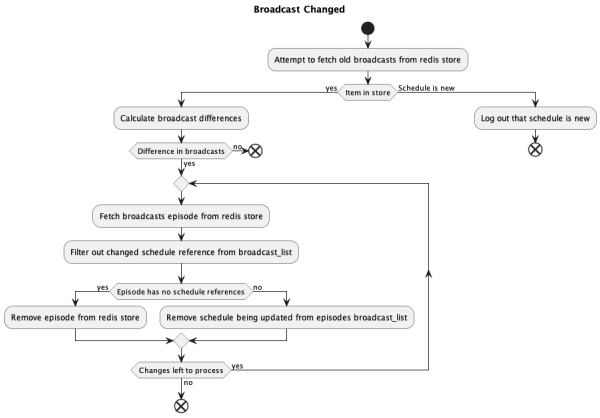


Figure 28: Activity diagram showing logic to handle broadcast episode change.

5.4.1.2 Schedule Deletes

Schedule deletes are one of the simpler events. Other than some basic guards to check objects exists in redis, all episode object referenced in the schedules broadcasts have the schedule being deleted removed from their broadcast list. If this list is empty then that episode is also removed from the schedule store. After all this is done, the schedule is also removed from the schedule store.

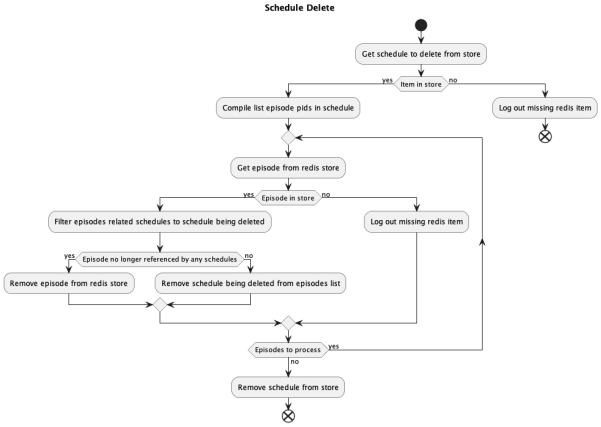


Figure 29: Activity diagram showing logic when schedule delete notification received.

5.4.1.3 Catalgoue Updates

Catalogue updates must also be processed by the schedule generator as they contain information about titling, programme descriptions and more.

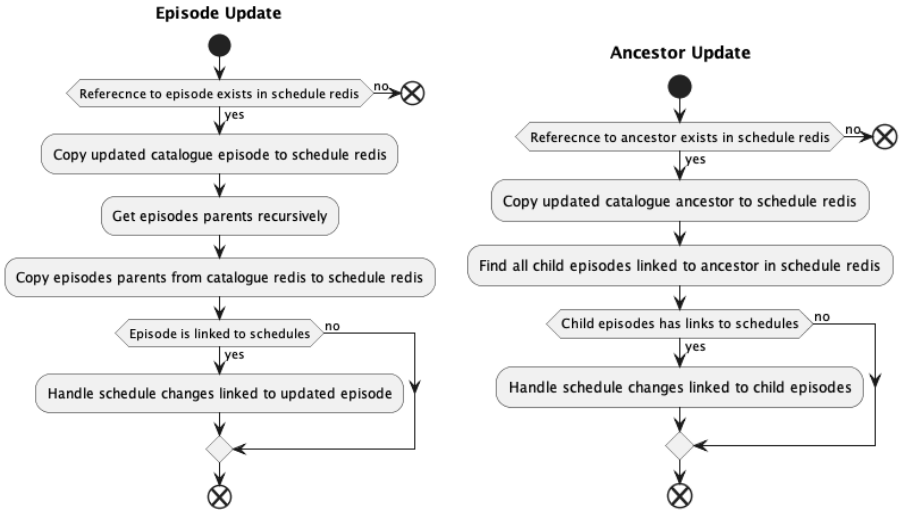


Figure 30: Activity diagrams showing logic when episode (left) and ancestor update (right) notifications received.

Episode and ancestor updates are similar. Episode updates must update themselves but also that episodes parents as the parents contain additional tilting information required for the schedule to be accurate.

Due to this titling information being held at multiple levels of the catalogue data, when an ancestor is updated it must update itself, but then also find all episodes in the schedule store that are dependant on it.

Episodes contain a list of schedules that reference it. These schedules must be updated, so a method was created that could handle these schedule changes in a batch form.

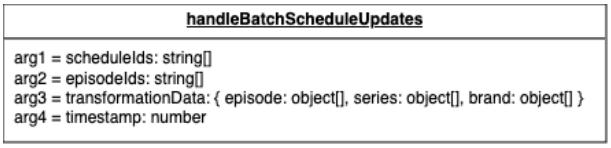


Figure 31: Class diagram showing the interface for the handling schedule updates triggered by catalogue notifications.

The method used the above interface and could be called the same way when updating an episode or an ancestor. The data includes:

- **scheduleIds** - Schedules in episodes broadcast lists.
- **episodeIds** - Episodes effected by the change, singular on episode update, multiple on ancestor update.

- **transformationData** - Episode/series/brand data needed to transform the all schedules.
- **timestamp** - Used for consistent update time on all updates.

Using this interface logic can then carried out to only update the broadcasts that have changed within a schedule instead of the entire schedule itself.

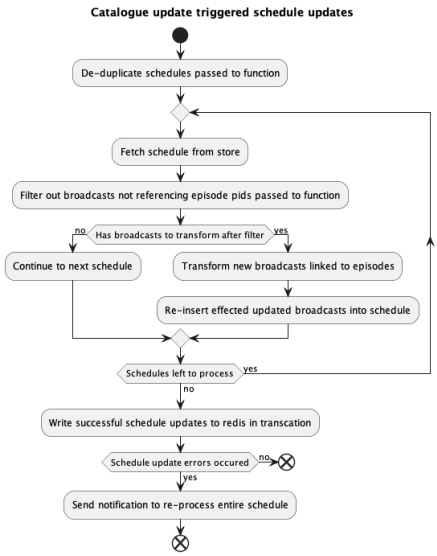


Figure 32: Activity diagram showing logic that happens when a catalogue notification triggers schedule updates.

5.4.1.4 Catalogue Deletes

It was determined during the spike that catalogue deletes would not need to be processed due to the creation of the garbage collector removing orphaned ancestor items, and schedule events taking care of episode removals. This saves a lot of additional lambda invokes over time.

5.4.2 Coldstarts

We use the term coldstart to refer to the refreshing of all data, in schedules terms all schedules will be checked to see if an update has occurred. This is not related to a cold start/boot in relation to serverless architectures (Microsoft Azure, 2018), although this is something that can affect our lambdas. We already had a coldstart in place but changes would have to be made to work with the new delta/notifications environment.

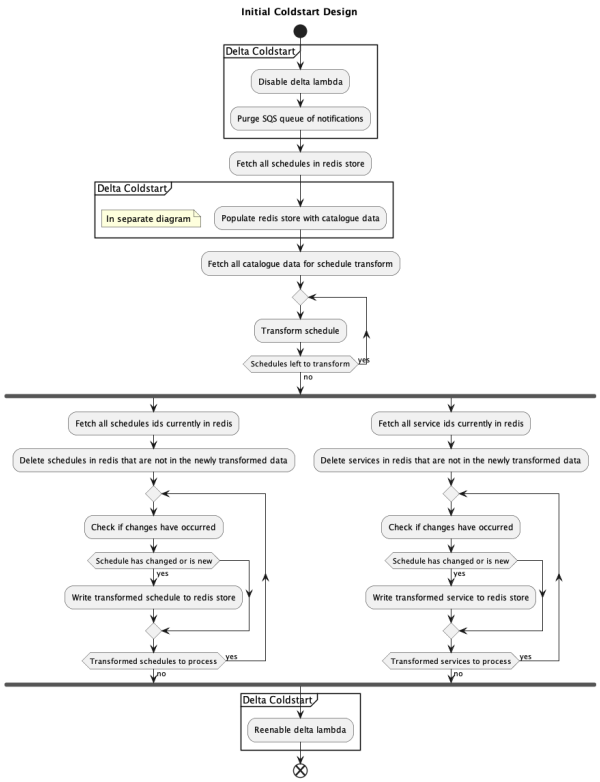


Figure 33: Activity diagram showing initial idea for coldstart changes.

The above diagram shows the initial design for how the coldstart would function. The changes seem relatively simple:

1. Disable triggering of lambda on delta notification event. This is so no changes can interfere with the refresh of data.

2. Purge the queue of updates before starting processing to stop unnecessary triggers of the lambda after the cold start has finished.

3. Populate the redis with all the catalogue data it needs, this is done on schedule updates also.

4. Re-enable the triggering of lambda on delta notification event.

Disabling and enabling of the lambda trigger can be achieved through the UpdateEventSourceMapping api (Amazon Web Services, 2024l) provided by AWS. At the start, the trigger can be disabled, then re-enabled once the cold-start is finished.

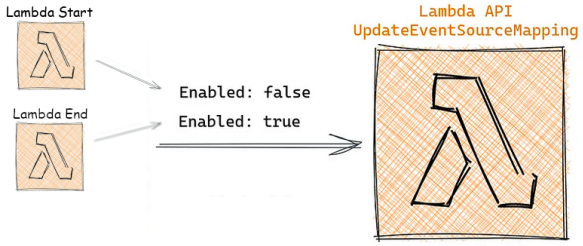


Figure 34: Diagram edited from Charles (2021) to show lambda mapping states.

Purging the SQS queue of delta notifications can also be done though a similar API and the population of redis is already done per schedule, so this seems like an easy switchover to work in the new world of delta notifications.

However, as work began the code grew more and more complex, due mainly to the new logic of copying data over into the new schedule redis keyspace. Maintaining the broadcast list in the episodes would mean making the population logic complex and hard to follow. Code complexity can be a huge problem with developers not wanting to edit the code in fear of breaking the current workings, difficulty to maintain (Mateus, Andre, 2022 and Olbrich et al, 2009) and the *truck/bus factor* which illustrates how hard it would be for a new team member to understand with no help from the creator (Avelino et al, 2016). In addition to this, a study done by (Taibi et al, 2017) found that *'Smells related to size and complexity are considered harmful by a higher percentage of participants than others.'*. The system created is complex and has multiple inputs that can interfere with each other, the same study also found that this higher complexity is less likely to be refactored in the future (Taibi et al, 2017).

Complexity	Level of occurrence (%)		
	Seldom	Regularly	Often
Low	7.94	23.81	57.14
Medium	12.70	55.56	19.05
High	46.03	28.57	14.29

For these reasons a new approach was thought of. The coldstart would tidy up the differences between it's *truth* the internal store, and it's output. Anything not in the internal store is to be removed with the rest of the schedules being processed as if it was a delta notification event.

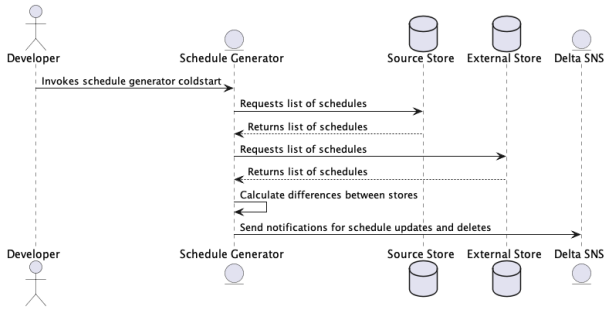


Figure 35: Sequence diagram showing the new flow of the final coldstart solution.

This allows the re-use of existing code whilst reducing the complexity and potential code duplication of the coldstart by a sizeable amount.

5.4.3 Garbage Collector

Nowadays garbage collectors are synonymous with programming languages such as Java (Xu et al, 2019), however the first language to implement garbage collection was LISP in the 1960s (Matam, 2023). Before then programmers had to manually assign memory for variables stored on the heap, as well as unassign the memory when they were finished using it (Fakhoury et al, 2024 and Matam, 2023). Garbage collection stops memory leaks as any unused memory is reclaimed (Microsoft, 2023).

Whilst doing the spike it was thought that a form of *garbage collector* may help simplify some of the logic around the removal of catalogue items from redis. Episode were easy to remove as once their associated broadcast list was empty they no longer needed to be stored. But due to having to traverse upwards to find other linked items, it became complex and time consuming to calculate if an episodes parents (series) and grandparents (brands) also needed to be removed from the schedule redis store.

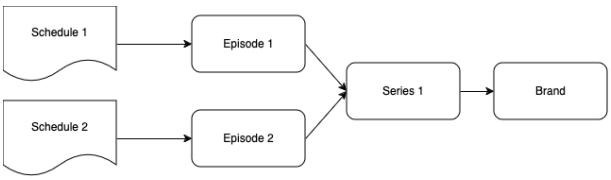


Figure 36: Example tree and associations between objects.

Looking at the figure above outlines one possible scenario. If *Episode 1* is removed from *Schedule 1* then the episode object should be removed from redis and so should it's parent *Series 1* as that also no longer has any schedule relations. However *Brand* must stay as it still has *Episode 2* and *Series 2* that depend on it and relate it to *Schedule 2*. Due to the potential to fan out like this, all data stored in the redis keyspace would have to be checked to see if there was any schedule still depending on the top level object (the brand). This takes a lot of memory and bandwidth to retrieve all this data. It was for that reason that it was decided, once a day a garbage collector should be ran to tidy up anything no longer required.

Figure 37: Garbage collector architecture.

Above is the final architecture, a lambda is triggered by a scheduler once a day a 3AM to tidy up the leftover series and brands. In a later **section** the idea of moving all catalogue removals, including episodes, to the garbage collector is explored.

5.4.4 End-to-end tests

In this section I want to discuss the larger tests that were created as part of the project. These include end-to-end (e2e) and comparison tests. As a team we follow the pattern of tests layed out in the test pyramid.

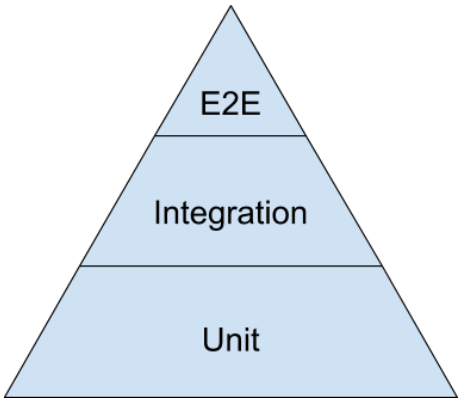


Figure 38: Test pyramid (Wacker, 2015).

Unit tests and integration tests are written alongside the code, and can run on project build as they are cheap and quick to run (Spinellis, 2017). E2e tests sit at the top of this pyramid and should be used sparingly. This is not always the case, in other types of development, for example mobile, this pyramid can be turned upside down (Knott, 2015, cited in Contan, Dehelean and Miclea, 2018). This is probably due to mobile application being primarily UI driven, which is the opposite for this project.

In addition to this e2e and integration tests often get confused with one another. Integration tests *'ensure synchronization between modules'* (Testim, 2021), and are more comparable to a unit test. Whereas a unit test is concerned on a single function/method an integration would test the calling of methods by other methods and ensure integration between these two functions are working as expected. E2e tests are focused on actions/events that the system must deal with as normal operation and could be described as automated manual testing (Testim, 2021).

Due to their higher complexity, e2e tests are not without their potential problems. These include maintenance the written tests, flakiness (are prone to randomly pass or fail without any discernable reason), time taken to run the tests and the cost of both the execution of the tests and the time to write them (Yang and Leotta et al, 2023).

One of the larger problems we've faced in the passed is asynchronicity between tests (Leotta et al, 2023). For example if we upload a schedule at the start of the pipeline and want to test it's structure at the end of the pipeline where partners would receive it, there's a lot that can interfere with expected outputs. For the most part these bugs are when tidy up of a previous test is

not carried out fully. If two test use the same identifier for a schedule and then don't remove said schedule, this can result in tests not running how expected. It's for this reason during the development of these new tests the code below was written to ensure that the schedule item about to be asserted on is not stored in the redis prior to the test starting.

```
1     waitFor(`Ensure ${schedule} is not present before running test
          `, async () => {
2       const bsd = await fetchFromRedis({
3         command: 'isAbsent',
4         keySpace: 'bsd_v2.0',
5         sidDates: [schedule],
6       })
7       expect(bsd).toEqual([null])
8     })
```

Listing 1: Code to ensure schedule to asserted on is not present at the start of the test.

Our current tests are written in *scenarios* by our test team, a list of which can be seen in **Appendix G**. Currently these cover all possible scenarios, however have a lot of overlap and repetitiveness. In an ideal world a test would cover an individual use case (Daly, 2022). These 14 scenarios could then be a part of these use case tests which would speed how quickly they run, thus lowering the cost. These use cases could be the following:

- Update a schedule
- Delete a schedule
- Update an episode
- Update an ancestor (series/brand)
- Tidy up orphaned items in redis (garbage collector)

This lowers the number of running test by 9 and removes a lot of time and duplication taken by setting up the scenarios. Due to time constraints and prioritisation of other projects for the future, this refactor has not yet been undertaken but is in the backlog of things to do in the future. The cost saved from changing this wouldn't be high, and doesn't justify the time it would take to refactor.

For this project we stuck with the scenarios, some had already been done in previous work which meant we only had tests to write for the new notification functionality. After discussion with the test team these tests include:

1. Hydration of episode stub to full episode object on episode update, as well as the episode parents and grandparents being transferred. This test would also make sure any schedules associated with the episodes titling was updated.

2. Complex logic on schedule update, when a broadcasts episode changes in the schedule, if that episode has no remaining links to any other schedules it should be removed from the schedule redis.

3. Test whether correct supplementary data matches what is in the schedule. For partner requests we also send episode/series/brand data, this data should be relevant to the broadcasts contained within the schedule.

There were originally more test scenarios to be done as part of this project. However after discussing with both developers and testers it was decided that these were either already being tested by unit tests or other parts of the testing library, or were unnecessary. This work was removed from the board which can be seen in the burn-up charts in the **Outputs** section.

Finally we created comparison tests, to compare the old static output and new delta/notification output. These tests could be compared to extract, transform, load (ETL) testing (Talend Inc, 2024). These tests will give us the confidence to switch our partners over to the new system. We decided upon a 1 week time frame where these comparisons tests must pass continuously before we are happy to make the switch. On failure the logs from the test will help us debug any current and future issues with the system.

5.5 Release

The release stage is the final stage in our workflow. This is when software gets released to the live environment and becomes available to partners.

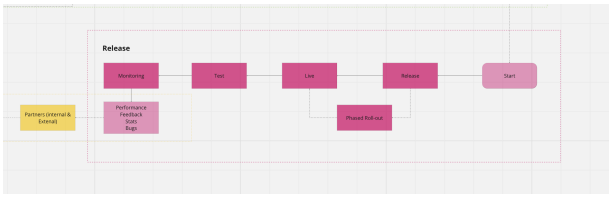


Figure 39: Release stage of our ways of working.

However in this case it doesn't become available to partners until we have confidence in the new system. We have config for each partner that specifies what data they have access to. In the case of schedules, this includes keyspace information on where the data is retrieved from. As previously discussed in the **Research** section, the new and old system will run in parallel until our comparison tests convince us that the new system outputs the same as the old.

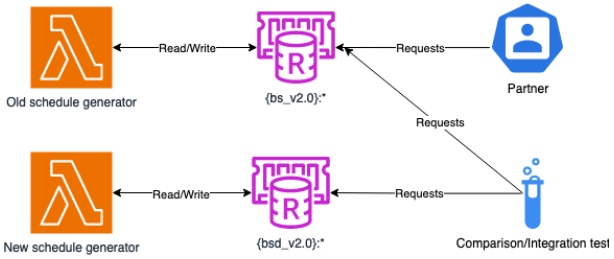


Figure 40: Diagram showing access to keyspaces.

Until then partners will still access data from the old system, with a simple config change being the rollover/rollback mechanism. If we wanted we could setup an automatic rollback and huddle system (Sathre, Zambreno, 2008), with the huddle portion consisting of developers being called out if outside of working hours to monitor and remedy the situation. This can be done in many ways but one way would be through AWS alarm actions (Amazon Web Services, 2024k). An alarm going into error would triggers a rollback of the configs version in S3 (Amazon Web Services, 2024d).

Figure 41: Automatic rollback architecture on alarm error.

For our case this is most likely over-engineering and unnecessary. A config change only takes a minute to do and if there ever was an error we would be notified and called out to fix the problem when necessary.

Once the switchover happens we wait for partner feedback and use created dashboards, these will be shown in the next section, to monitor the new software. Alongside this, our comparison test and side-by-side running of new and old systems will continue for a short while after release, allowing an easy rollback to the old system. Next this old lambdas scheduler will be disabled, but the lambda itself kept in case of an emergency. Finally, after sufficient time has passed the old system can be deconstructed and removed completely.

6 Outputs

In this penultimate section, I will share some of the outputs the project has delivered. I will use these to reflect on the different parts of the project each output is associated with. To do this reflection I will use the Experience, Reflect, Action (ESA) framework founded by Melanie Jasper (2003, p.2).

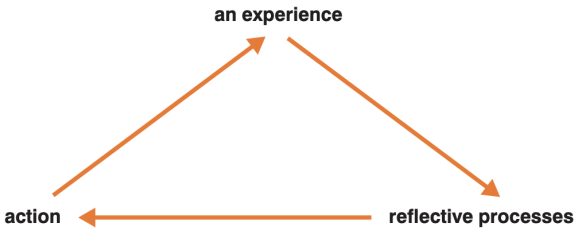


Figure 42: ERA reflection model founded by Jasper (2003, p.2).

6.1 Burn-up Charts

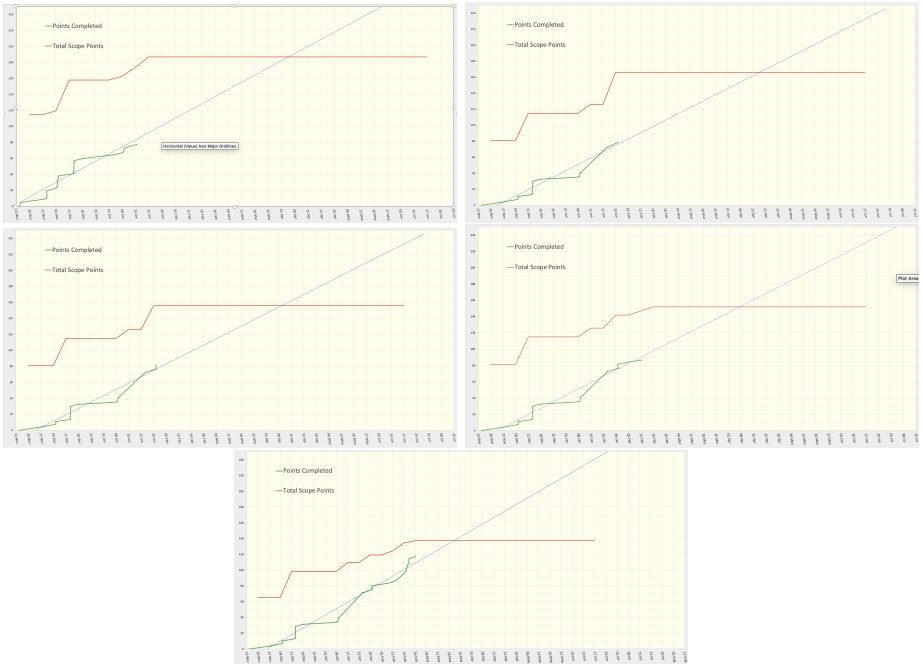


Figure 43: Burn-up charts throughout the project.

Experience - Reflection - Action -

6.2 Final System Architecture

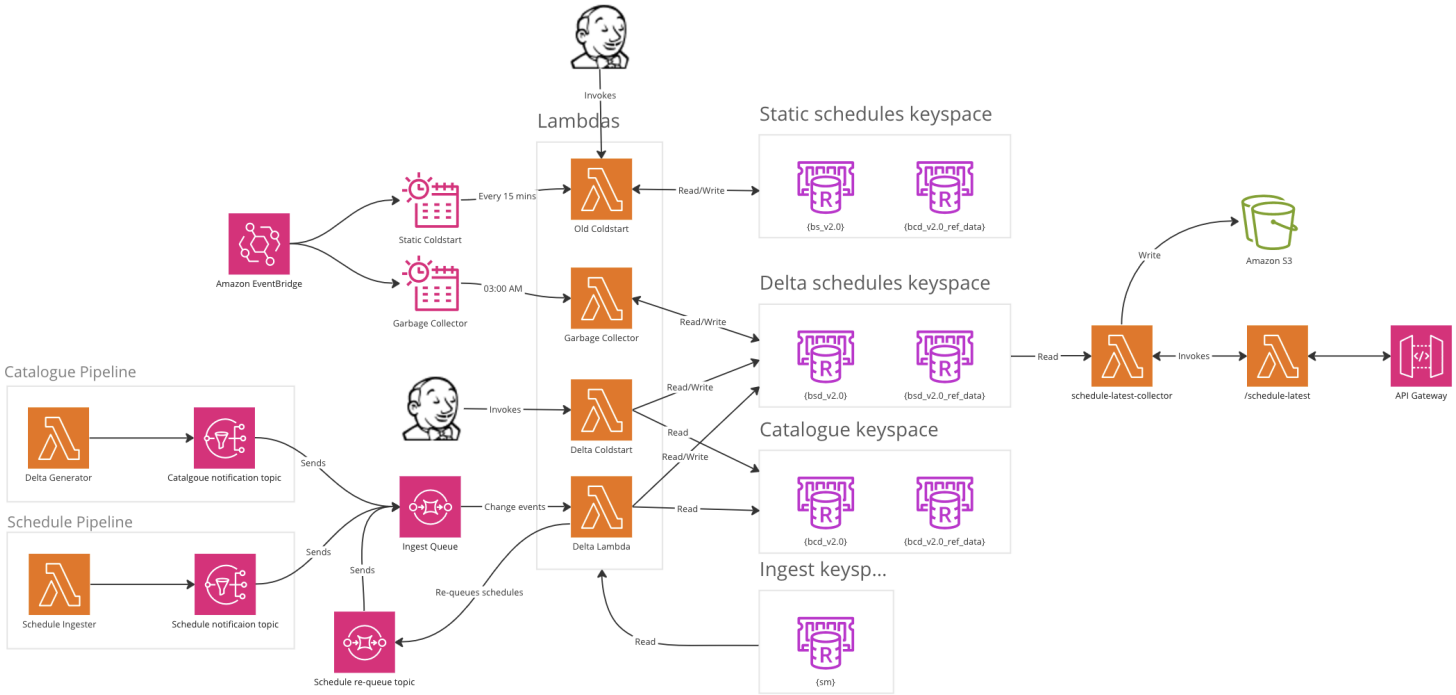


Figure 44: Final architecture for project.

Experience - Reflection - Action -

6.3 Dashboard

Experience - Dashboards have been created using a tool called Grafana (2024) that pulls metrics () from AWS and displays them all in one place. This allows us to check for anomalies and errors. Some metrics such as lambda run-times and invocations are provided by AWS but custom metrics have also been created to track errors, redis writes and schedule re-queues. Metrics are collected during the run time of the lambda using variations of the following code.

```
1     monitoring.collect({
2       typeId: 'Processing',
3       result: 'throughput',
4       dimensions: {
5         ObjType: 'service_schedule',
6         EventType: 'Delete'
7       },
8       value: 1
9     })
```

Listing 2: Code used to update a metric this variation tracks a schedule delete.

Reflection - I've used metrics and dashboards throughout my time at the BBC so this is not something that is new to me. Usually the process of adding these metrics is done at the very end of development. This stops monitoring during development which is only a negative. Due to the use of a spike in this project we were able to know what metrics we wanted to track from the start and therefore added them along the way. This helped a lot as metrics on both test and live could be compared to one another as a way to validate that changes had occurred.

Action - Continue to use spikes where necessary and put more thought into what kind of metrics we want to track before development starts. This allows comparisons between different dashboards as well as saves one or multiple extra tasks at the end of development to create the metrics and dashboards.

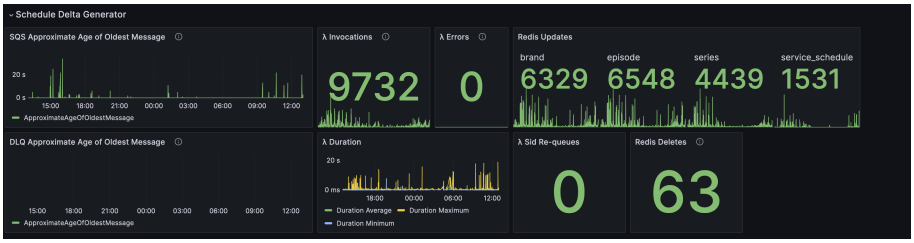


Figure 45: Created dashboard in Grafana.

6.4 Code Base and Commits

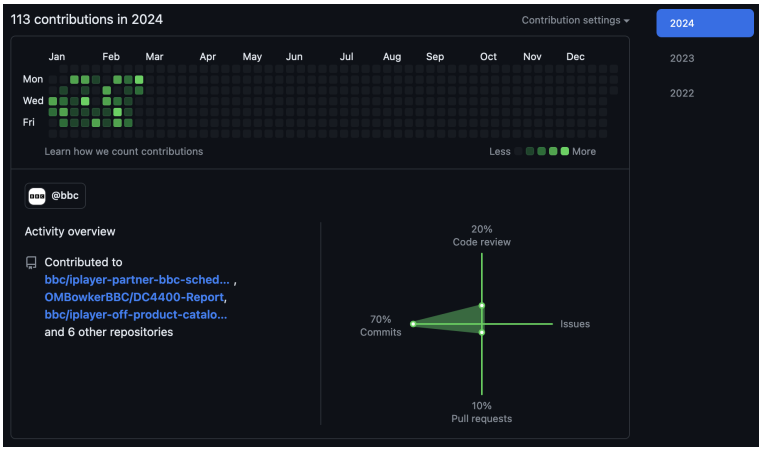


Figure 46: Github contributions for my work profile.

Experience - Reflection - Action -

7 Future Work and Conclusions

This report has discussed the creation and upgrade of an enterprise system that will be used by millions of people across the UK.

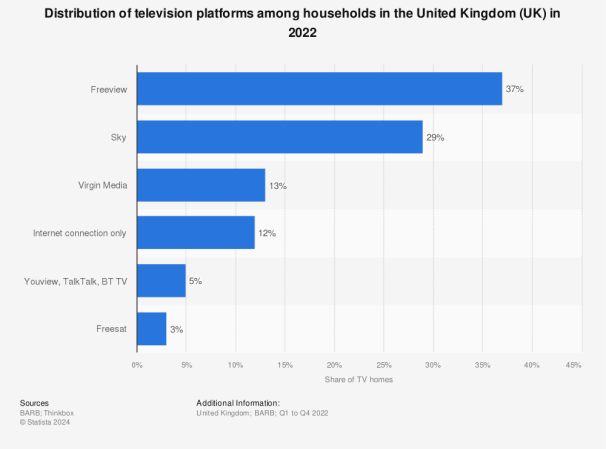


Figure 47: Chart showing freeview, a partner using the schedule feed, having the highest usage in UK homes (Statista, 2022).

The report covered all parts of the software development life cycle (SDLC), including requirements, business drivers for the project, research, a spike of the initial design, creation of tasks and work slices for a development and test team to work on, the creation of the software and finally, the releasing of this new software to a live system depended on by partners.

To finish off this report I will discuss three future upgrades that could be made to the schedules system to improve both its timeliness of data updates and a way that the system can be parallelised and simplified.

7.1 Notifications direct to partners

The new system relies on partners requesting data in order to update their UIs. This can mean that partners interfaces are still out of date, even though our internal store is as up to date as possible.

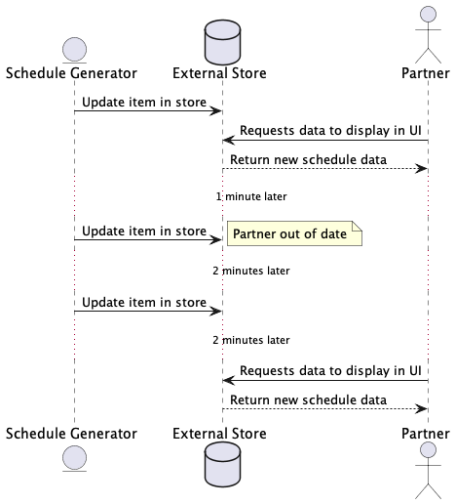


Figure 48: Sequence diagram showing how partners become out of date in new system.

In addition to this, re-requesting schedules that haven't been updated is a waste of time and costs the organisation money for data transfers (Amazon Web Services, 2024g). Below is an architecture that attempts to solve this problem by directly sending notifications to partners when an update occurs.

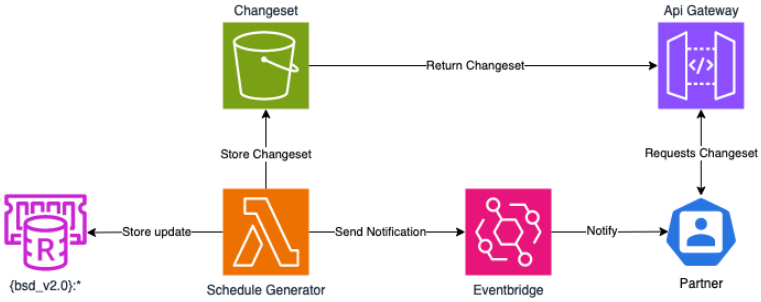


Figure 49: Potential architecture to serve notifications to partners.

A notification ID is sent to a partners system, they then request a *'changeset'* from our API which consists of only the data that has been updated. This new schedule can then be used instead of the old one, without doing a full refresh of all the schedules a client wants access to.

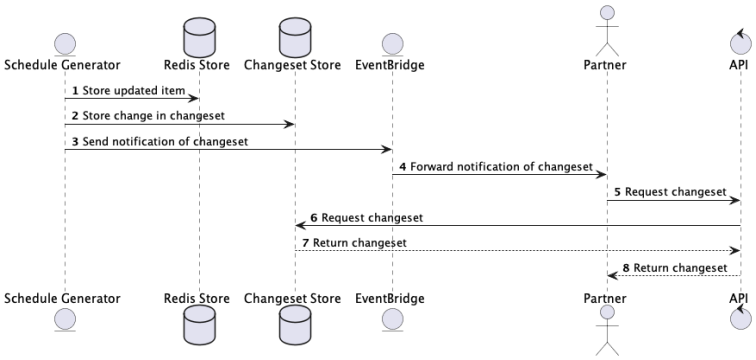


Figure 50: How an schedules can be kept up to date using changesets.

7.2 Parallelise the current system

The new system is single-threaded and is unable to run in parallel due to shared memory between the threads needing to be managed. The piece of memory at fault here is the broadcast list held in episodes that link that episode to schedules in which it is referenced. The main problem here is redis can't help us with any form of locking, unless we implement it ourselves, more on this can be found in the **Research** section of this report.

During this research DynamoDB came up as a possible solution due to it's use of optimistic locking capabilities (Kanungo, Morena, 2023 and Amazon Web Services 2024h). Using a column to monitor for changes, code could be written using Amazons Software Development Kit (SDK) (IBM Cloud Education, 2021) to ensure that no changes had occurred to the underlying before writing to the table.

ID	Associations	Version
Unique identifier of of the object which would be used for quick lookups.	For series and brand objects this would be a list of children (series/episode) that the object refers to. Foe episodes this would now be where the broadcast list of related schedules is stored.	Number that is used to determine if changes have occurred using optimistic locking. This will be incremented by 1 every time a change happens.

Table 3: Example of dynamoDB table columns that could be used.

This table can be combined with the following SDK command to optimistically lock the data being edited. In this code *ConditionExpression* is the argument that determines what row is locking the record.

```
1   const updateCommand = new UpdateItemCommand({
2     TableName: 'schedule-associations',
3     Key: marshall({ id: objectIdToUpdate }),
4     UpdateExpression: 'set associations :newAssociations',
5     ConditionExpression: 'version = :previousVersion',
6     ExpressionAttributeValues: marshall({
7       ':newAssociations': newAssociations,
8       ':previousVersion': initialObject.version
9     })
10  })
```

Listing 3: SDK command sent to optimistically lock writes to the assocaitions column.

Finally this can all be integrated into a new architecture that is outlined below.

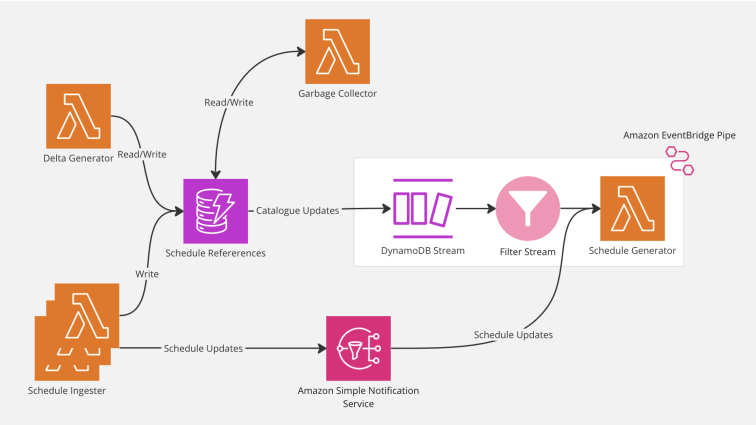


Figure 51: Option for parallelised architecture.

The system still takes input from both the schedule ingester and catalogue pipeline as before. However the aforementioned dynamoDB table is populated by ingester and the catalogue pipeline, with the schedule generator no longer needing access to that data. Instead an AWS pipe (Amazon Web Services, 2024i) is used to trigger invocation of the delta lambda for catalogue updates, whilst schedule updates function the same. AWS pipes consist of 4 sections:

1. **Source** - DynamoDB streams can be used as a source to a pipe (Amazon Web Services, 2024j), this stream representing a change to the table. This event can contain both old and new items from the dynamoDB table, as well as the type of event that it is (Silveria, 2020).

2. **Filter** - We can filter out events based on the type of change made, we only want to trigger a lambda invocation on a MODIFY and not on an INSERT or DELETE as these events are covered by the standard schedule processing.

3. **Enrichment** - Not used in this example but can trigger extra enrichment of the data prior to the final target.

4. **Target** - This is where to send the filtered data to. In this case it's the schedule generator itself.

The inclusion of the dynamoDB table allows the use of optimistic locking to eventually parallelise the entire schedule pipeline. However there are other huge benefits here as well with the highlights being:

- There is no longer a need to copy over catalogue data to the schedules redis store as it was only used to maintain the link between episode and schedules via the broadcast list. This lowers the complexity of the schedule generator and allows the possibility for parallelism.

- The catalogue pipeline can be made to only send updates for objects that exist in schedules.

```
1          const updateCommand = new UpdateItemCommand({
2            TableName: 'schedule-associations',
3            Key: marshall({ id: updatedCatalogueItem.id }),
4            UpdateExpression: `set version ${initialObject.
                 version + 1}`,
5            ConditionExpression: `id = ${updatedCatalogueItem.id
                 }`,
6          })
```

Listing 4: SDK command sent by catalogue pipeline to ignore non schedule related catalogue items.

The above code once again uses the *ConditionExpression*, but instead of optimistically locking here it is used to check if the objects exists. This works as if the object doesn't exists in the table it can't have an id. This will stop needless invocations of the schedule generator.

- An additional refinement stage could be added to the pipe to check that the modification to data that has happened would cause a change to a schedule. Only certain fields changes in catalogue items cause this so this could again save more invocations.

Figure 52: AWS pipe including lambda enrichment stage.

7.3 Garbage Collection Consolidation

A small change that could be made is for the garbage collector alone to handle removal of catalogue items, with the schedule generator only removing schedules. This makes a lot of sense especially when applying the architectural principle of separation of concerns (SoC) which *'asserts that software should be separated based on the kinds of work it performs'* (Smith, 2023, ch. 3, p. 11). In addition to adhering to well known principles it would also simplify what has become a bloated and complex schedule generator.

The initial reason for removing the episode immediately upon deletion of schedule or the realisation that an episode was no longer referenced in any schedule was for timeliness for partners. However this doesn't make sense, partners would never receive this orphaned information anyway due to how the data is collected for them on request.

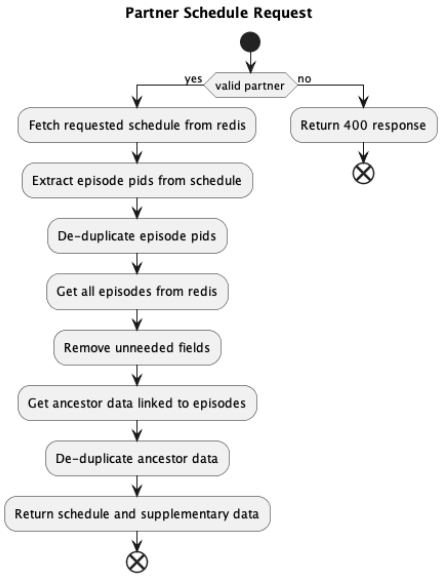


Figure 53: How supplementary catalogue data is calculated for partner request.

The benefits of removing this make things much clearer and is a quick win in comparison to some of the other suggestions for future work.

7.4 Conclusions

8 References

BBC. (2022) *Timeline - History of the BBC* [online]. Available at https://www.bbc.co.uk/historyofthebbc/bbc-100/timeline/ (accessed on 18th January 2024).

Pilnick, C. Baer, S, W. (1973) *Cable television: A guide to the technology* [online]. Available at https://www.rand.org/content/dam/rand/pubs/reports/2006/R1141.pdf (accessed on 18th January 2024).

Ofcom. (2023) *Media Nations report 2023* [online]. Available at https://www.ofcom.org.uk/__data/assets/pdf_file/0029/265376/media-nations-report-2023.pdf (accessed on 18th January 2024).

Ampere Analysis. (2023) *Almost half of Internet users say they have switched off broadcast TV* [online]. Available at https://www.ampereanalysis.com/press/release/dl/almost-half-of-internet-users-say-they-have-switched-off-broadcast-tv (accessed on 18th January 2024).

Statista. (2023) *Household penetration of smart TV sets in the United Kingdom (UK) from 2014 to 2023* [online]. Available at https://www.statista.com/statistics/654074/smart-tv-penetration-in-uk-households/ (accessed on 18th January 2024).

Statista. (2022) *Distribution of television platforms among households in the United Kingdom (UK) in 2022* [online]. Available at https://www.statista.com/statistics/297494/top-uk-tv-platforms-among-households/ (accessed on 29th February 2024).

Amazon. (2021) *Linear TV Integration Guide Overview* [online]. Available at https://developer.amazon.com/docs/fire-tv/linear-tv-integration-guide-overview.html (accessed on 18th January 2024).

Davie, T. (2022) *A digital-first BBC* [online]. Available at https://www.bbc.co.uk/mediacentre/speeches/2022/digital-first-bbc-director-general-tim-davie (accessed on 18th January 2024).

Sparks, R. (2024) *OKRs: the ultimate guide to objectives and key results* [online]. Available at https://www.atlassian.com/agile/agile-at-scale/okr (accessed on 18th January 2024).

Bowker O. (2023) *Analysis of BBC process and proposal for a new API key management system.* Assignment for DC4000, MSc Digital and Technology Solutions Specialist, Aston University, Unpublished.

BBC Partnerships. (2023) *Partnerships - one year on.* BBC. Unpublished.

Somerville, I. (2016) *Software engineering.* Tenth edition, Global edition. London: Pearson Education.

Somerville, I. (2021) *Engineering Software Products: An Introduction to Modern Software Engineering.* Global edition. London: Pearson Education.

Jasper, M. (2003) *Beginning Reflective Practice*. Cheltenham: Nelson Thornes Ltd.

Amazon Web Services. (2024a) *AWS Lambda* [online]. Available at `https://aws.amazon.com/lambda/` (accessed on 2nd February 2024).

Amazon Web Services. (2024b) *Amazon Simple Notification Service* [online]. Available at `https://aws.amazon.com/sns/` (accessed on 2nd February 2024).

Amazon Web Services. (2024c) *Amazon Simple Queue Service* [online]. Available at `https://aws.amazon.com/sqs/` (accessed on 2nd February 2024).

Amazon Web Services. (2024d) *Amazon S3* [online]. Available at `https://aws.amazon.com/s3/` (accessed on 2nd February 2024).

Amazon Web Services. (2024e) *Amazon DynamoDB* [online]. Available at `https://aws.amazon.com/dynamodb/` (accessed on 2nd February 2024).

Amazon Web Services. (2024f) *Amazon ElastiCache* [online]. Available at `https://aws.amazon.com/elasticache/` (accessed on 9th February 2024).

Amazon Web Services. (2024g) *How AWS Pricing Works* [online]. Available at `https://docs.aws.amazon.com/pdfs/whitepapers/latest/how-aws-pricing-works/how-aws-pricing-works.pdf` (accessed on 20th February 2024).

Amazon Web Services. (2024h) *Optimistic locking with version number* [online]. Available at `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBMapper.OptimisticLocking.html` (accessed on 21st February 2024).

Amazon Web Services. (2024i) *Amazon EventBridge Pipes* [online]. Available at `https://aws.amazon.com/eventbridge/pipes/` (accessed on 20th February 2024).

Amazon Web Services. (2024j) *Amazon DynamoDB stream as a source* [online]. Available at `https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-pipes-dynamodb.html` (accessed on 21st February 2024).

Amazon Web Services. (2024k) *PutMetricAlarm* [online]. Available at `https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/API_PutMetricAlarm.html#API_PutMetricAlarm_Errors` (accessed on 19th February 2024).

Amazon Web Services. (2024l) *UpdateEventSourceMapping* [online]. Available at `https://docs.aws.amazon.com/lambda/latest/api/API_UpdateEventSourceMapping.html` (accessed on 23rd February 2024).

IBM. (2024) *What is Redis?* [online]. Available at `https://www.ibm.com/topics/redis` (accessed on 9th February 2024).

IBM Cloud Education. (2021) *SDK vs. API: What's the Difference?* [online]. Available at `https://www.ibm.com/blog/sdk-vs-api/` (accessed on 21st February 2024).

Redis Ltd. (2024a) *Redis FAQ* [online]. Available at https://redis.io/docs/get-started/faq/ (accessed on 9th February 2024).

Redis Ltd. (2024b) *Redis pipelining* [online]. Available at https://redis.io/docs/manual/pipelining/ (accessed on 9th February 2024).

Redis Ltd. (2024c) *Transactions* [online]. Available at https://redis.io/docs/interact/transactions/ (accessed on 9th February 2024).

Redis Ltd. (2024d) *Redis modules and blocking commands* [online]. Available at https://redis.io/docs/reference/modules/modules-blocking-ops/ (accessed on 9th February 2024).

Oracle Corporation. (2010) *Defining Multithreading Terms* [online]. Available at https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html (accessed on 9th February 2024).

Eyng, R. (2019) *Redis: Pipelining, Transactions and Lua Scripts* [online]. Available at https://rafaeleyng.github.io/redis-pipelining-transactions-and-lua-scripts (accessed on 9th February 2024).

Visual Paradigm. (2024) *What is Spike in Scrum?* [online]. Available at https://www.visual-paradigm.com/scrum/what-is-scrum-spike/ (accessed on 9th February 2024).

Ugarte, A. (2024) *What Is Thread-Safety and How to Achieve It?* [online]. Available at https://www.baeldung.com/java-thread-safety (accessed on 9th February 2024).

Graefe, G. (2016) *Revisiting optimistic and pessimistic concurrency control* [online]. Available at https://www.labs.hpe.com/techreports/2016/HPE-2016-47.pdf (accessed on 9th February 2024).

Thornton, G. (2001) *Optimistic Locking with Concurrency in Oracle* [online]. Available at https://www.orafaq.com/papers/locking.pdf (accessed on 9th February 2024).

Weston, T. (2011) *Optimistic and Pessimistic Concurrency Control with Shared Memory Models* [online]. Available at https://baddotrobot.com/resources/concurrency-control-1.0-draft.pdf (accessed on 9th February 2024).

Apache Software Corporation. (2013) *Deadlocks* [online]. Available at https://db.apache.org/derby/docs/10.0/manuals/develop/develop75.html (accessed on 9th February 2024).

Heil, C. (2022) *Is Kanban the better Agile Approach in a Highly-Regulated Environment?* [online]. Available at https://www2.deloitte.com/content/dam/Deloitte/de/Documents/risk/Deloitte_Agile_Approach_in_Highly_Regulated_Environments.pdf (accessed on 12th February 2024).

Rehkopf, M. (2024) *Kanban vs. scrum: which agile are you?* [online]. Available at https://www.atlassian.com/agile/kanban/kanban-vs-scrum (accessed on

12th February 2024).

Mauvius Group Inc. (2021) *The official guide to the kanban method* [online]. Available at https://resources.kanban.university/wp-content/uploads/2021/06/The-Official-Kanban-Guide_A4.pdf (accessed on 12th February 2024).

Jenkins. (2024) *Pipeline Syntax* [online]. Available at https://www.jenkins.io/doc/book/pipeline/syntax/ (accessed on 12th February 2024).

Rodriguez, P. Haghighatkhah, A. Lwakatare, L, E. Teppola, S. Suomalainen, T. Eskeli, J. Karvonen, T. Kuvaja, P. Verner, J, M. Oivo, M. (2017) 'Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study', Journal of Systems and Software, Vol. 123, pp. 263-291. Available at: https://doi.org/10.1016/j.jss.2015.12.015

Taibi, D. Janes, A. Lenarduzzi, V. (2017) 'How developers perceive smells in source code: A replicated study', Information and Software Technology, Vol. 92, pp. 223-235. Available at: https://doi.org/10.1016/j.infsof.2017.08.008

Xu, L. Guo, T. Dou, W. Wang, W. Wei, J. (2019) 'An experimental evaluation of garbage collectors on big data applications', Proceedings of the VLDB Endowment, Volume 12, Issue 5, pp 570-583. Available at: https://doi.org/10.14778/3303753.3303762

Olbrich, S. Cruzes, S, D. Basili, V. Zazworka, N. (2009) 'The evolution and impact of code smells: A case study of two open source systems', 2009 3rd International Symposium on Empirical Software Engineering and Measurement. Available at: https://doi.org/10.1109/ESEM.2009.5314231

Mateus, L. Andre, H. (2022) 'How and why we end up with complex methods: a multi-language study', Empirical Software Engineering, Vol. 27, Issue 5, Article Number 115. Available at: https://doi.org/10.1007/s10664-022-10144-3

Avelino, G. Passos, L. Hora, A. Valente, T, M. (2016) 'A novel approach for estimating Truck Factors', 2016 IEEE 24th International Conference on Program Comprehension (ICPC). Available at: https://doi.org/10.1109/ICPC.2016.7503718

Department of Defence. (2021) *DoD Enterprise DevSecOps Fundamentals* [online]. Available at https://dodcio.defense.gov/Portals/0/Documents/Library/DoDEnterpriseDevSecOpsFundamentals.pdf (accessed on 14th February 2024).

NSA. (2023) *Defending Continuous Integration/Continuous Delivery (CI/CD) Environments* [online]. Available at https://media.defense.gov/2023/Jun/28/2003249466/-1/-1/0/CSI_DEFENDING_CI_CD_ENVIRONMENTS.PDF (accessed on 14th February 2024).

OWASP Foundation. (2023) *OWASP Top 10 CI/CD Security Risks* [online].

Available at https://owasp.org/www-project-top-10-ci-cd-security-risks/ (accessed on 14th February 2024).

Wikström, A. (2019) *Benefits and challenges of Continuous Integration and Delivery - A Case Study* [online]. Available at https://core.ac.uk/download/pdf/226768285.pdf (accessed on 14th February 2024).

Fairbanks, J. Tharigonda, A. Eisty, N, U. (2023) *Analyzing the Effects of CI/CD on Open Source Repositories in GitHub and GitLab* [online]. Available at https://arxiv.org/pdf/2303.16393.pdf (accessed on 14th February 2024).

GoRetro. (2023) *Ways of Working* [online]. Available at https://www.goretro.ai/glossary/ways-of-working (accessed on 14th February 2024).

Banerjee, A. (2017) 'System Changeover Policy and Precautions', International Journal of Advanced Engineering and Management, Vol. 2, No. 9, pp. 214-218. Available at: https://doi.org/10.24999/IJOAEM/02090048

Smyth, D. (2020) *Changeover Techniques* [online]. Available at https://smallbusiness.chron.com/changeover-techniques-34890.html (accessed on 15th February 2024).

IoRedis. (2024) *README - ioredis* [online]. Available at https://ioredis.readthedocs.io/en/stable/README/#transparent-key-prefixing (accessed on 15th February 2024).

Rustagi, V. (2023) *Strategies To Run Old & new Systems Simultaneously Using The Same Database* [online]. Available at https://metaorangedigital.com/blog/strategies-to-run-old-new-systems-simultaneously-using-the-same-database/ (accessed on 15th February 2024).

Bigelow, J, S. (2020) *What are the types of requirements in software engineering?* [online]. Available at https://www.techtarget.com/searchsoftwarequality/answer/What-are-requirements-types (accessed on 16th February 2024).

Karolis, N, Saulius, G. (2023) 'Causal Knowledge Modelling for Agile Development of Enterprise Application Systems', Informatica (Netherlands), Vol. 34 Issue 1, p 124. Available at: https://doi.org/10.15388/23-INFOR510

Eduardo, M. (2022) 'Moscow Rules: A Quantitative Exposé', Lecture Notes in Business Information Processing, Vol. 445, pp 19-34. Available at: https://doi.org/10.1007/978-3-031-08169-9_2

Kim, M. Park, S. Sugumaran, V. Yang, H. (2007) 'Managing requirements conflicts in software product lines: A goal and scenario based approach', Data & Knowledge Engineering, Vol. 61 Issue 3, pp 417-432. Available at: https://doi.org/10.1016/j.datak.2006.06.009

Kanungo, S, Morena, D, R. (2023) 'Concurrency versus consistency in NoSQL databases', Journal of Autonomous Intelligence, Vol. 7 Issue 3. Available at: https://doi.org/10.32629/jai.v7i3.936

Sanders, R. (1987), 'THE PARETO PRINCIPLE: ITS USE AND ABUSE', Journal of Services Marketing, Vol. 1 No. 2, pp. 37-40. Available at: https://doi.org/10.1108/eb024706

Hashimi, A, L. Gravell, A. (2020), 'Spikes in Agile Software Development: An Empirical Study', 2020 International Conference on Computational Science and Computational Intelligence (CSCI). Available at: https://doi.org/10.1109/CSCI51800.2020.00319

Hashimi, A, L. Abduldaem, A. Gravell, A. (2022), 'Common Spikes Success Factors: An Industrial Investigation within Agile Software Development', 2022 12th International Conference on Software Technology and Engineering (ICSTE). Available at: https://doi.org/10.1109/ICSTE57415.2022.00008

Hashimi, A, L. Gravell, A. (2019) 'A Critical Review of the Use of Spikes in Agile Software Development', The Fourteenth International Conference on Software Engineering Advances. Available at: https://www.researchgate.net/publication/340446921_A_Critical_Review_of_the_Use_of_Spikes_in_Agile_Software_Development

Spinellis, D. (2019) 'State-of-the-Art Software Testing', IEEE Software, Vol. 34, Issue 5, pp 4-6. Available at: https://doi.org/10.1109/MS.2017.3571564

Kruchten, P. Nord, L, R. Ozkaya, I. Visser, J. (2012) 'Technical Debt in Software Development: from Metaphor to Theory Report on the Third International Workshop on Managing Technical Debt', ACM SIGSOFT Software Engineering Notes Advances, Vol. 37, No. 5, p. 36. Available at: https://doi.org/10.1145/2347696.2347698

Sathre, J, Zambreno, J. (2008) 'Automated software attack recovery using rollback and huddle', Des Autom Embed Syst, Vol. 12, pp. 243-260. Available at: https://doi.org/10.1007/s10617-008-9020-4

Leotta, M. García, B. Ricca, F. Whitehead, J. (2023) 'Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey', 2023 IEEE Conference on Software Testing, Verification and Validation (ICST). Available at: https://doi.org/10.1109/ICST57152.2023.00039

Zheng, J. (2021) 'Research on software development and test environment automation based on association rules', 2021 IEEE International Conference on Emergency Science and Information Technology (ICESIT). Available at: https://doi.org/10.1109/ICESIT53460.2021.9696910

Burghate, N. (2018) 'Work Breakdown Structure: Simplifying Project Management', International Journal of Commerce and Management Studies (IJCAMS), Vol. 3, No. 2. Available at: https://www.ijcams.com/wp-content/uploads/2018/06/WBS.pdf

Contan, A. Dehelean, C. Miclea, L. (2018) 'Test automation pyramid from theory to practice', 2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR). Available at: https://doi.org/10.1109/

AQTR.2018.8402699

Glas, T. Hedén, F. (2021) *Managing Technical Debt in XP Teams* [online]. Available at https://fileadmin.cs.lth.se/cs/Education/EDAN80/Reports/2021/GlasHeden.pdf (accessed on 19th February 2024).

Microsoft. (2021) *Technical Spike* [online]. Available at https://microsoft.github.io/code-with-engineering-playbook/design/design-reviews/recipes/technical-spike/ (accessed on 19th February 2024).

Microsoft. (2023) *Fundamentals of garbage collection* [online]. Available at https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals (accessed on 23rd February 2024).

Martins, J. (2023) *7 common causes of scope creep, and how to avoid them* [online]. Available at https://asana.com/resources/what-is-scope-creep (accessed on 19th February 2024).

Raeburn, A. (2024) *The work breakdown structure (WBS) for project management: What it is and how to use it* [online]. Available at https://asana.com/resources/work-breakdown-structure (accessed on 19th February 2024).

The Open Group. (2016) *Implementation and Migration Elements* [online]. Available at https://pubs.opengroup.org/architecture/archimate30-doc/chap13.html#_Toc451758082 (accessed on 19th February 2024).

Jonkers, H. Proper, E. Lankhorst, M, M. Quartel, A, C, D. Iacob, M. (2011) 'ArchiMate(R) for Integrated Modelling Throughout the Architecture Development and Implementation Cycle', 2011 IEEE 13th Conference on Commerce and Enterprise Computing. Available at: https://doi.org/10.1109/CEC.2011.52

Fakhoury, R. Braginsky, A. Keidar, I. Zuriel, Y. (2024) 'Nova: Safe Off-Heap Memory Allocation and Reclamation', Leibniz International Proceedings in Informatics, Volume 286, Article number 15. Available at: https://doi.org/10.4230/LIPIcs.OPODIS.2023.15

Silveria, A. (2020) *GraphQL Live Querying with DynamoDB* [online]. Available at https://arxiv.org/pdf/2008.00129.pdf (accessed on 21st February 2024).

Wiggins, A. (2017) *The Twelve-Factor App* [online]. Available at https://12factor.net/ (accessed on 23rd February 2024).

Microsoft Azure. (2018) *Understanding serverless cold start* [online]. Available at https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/ (accessed on 23rd February 2024).

Charles, Z. (2021) *How to Enable/Disable a Lambda Trigger on a Schedule* [online]. Available at https://zaccharles.medium.com/how-to-enable-disable-a-lambda-trigger-on-a-sc (accessed on 23rd February 2024).

Matam, S. (2023) *The History of Garbage Collection* [online]. Available at https://www.linkedin.com/pulse/history-garbage-collection-sai-matam (accessed on 23rd February 2024).

Wacker, M. (2015) *Just Say No to More End-to-End Tests* [online]. Available at https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (accessed on 26th February 2024).

Daly, N. (2022) *What Is a Use Case?* [online]. Available at https://www.wrike.com/blog/what-is-a-use-case/ (accessed on 26th February 2024).

Testim. (2021) *End-to-End Testing vs Integration Testing* [online]. Available at https://www.testim.io/blog/end-to-end-testing-vs-integration-testing/ (accessed on 26th February 2024).

Yang, L. (no date) *Factors to Consider When Implementing Automated Software Testing* [online]. Available at https://apps.dtic.mil/sti/tr/pdf/AD1022569.pdf (accessed on 26th February 2024).

Talend Inc. (2024) *ETL testing: A comprehensive guide to ensuring data quality and integration* [online]. Available at https://www.talend.com/resources/etl-testing/ (accessed on 26th February 2024).

Grafana Labs. (2024) *Grafana: The open observability platform — Grafana Labs* [online]. Available at https://grafana.com/ (accessed on 27th February 2024).

Smith, S. (2023) *Architect Modern Web Applications with ASP.NET Core and Azure* [online]. Available at https://raw.githubusercontent.com/dotnet-architecture/eBooks/main/current/architecting-modern-web-apps-azure/Architecting-Modern-Web-Applications-NET-Core-and-Azure.pdf (accessed on 26th February 2024).

Lloyd, K. (2023) *iPlayer Schedules - BBC Partner Schedule Generator - Delta Update Technical Design*. BBC. Unpublished.

9 Appendix

9.1 Appendix A - Partnerships Objectives

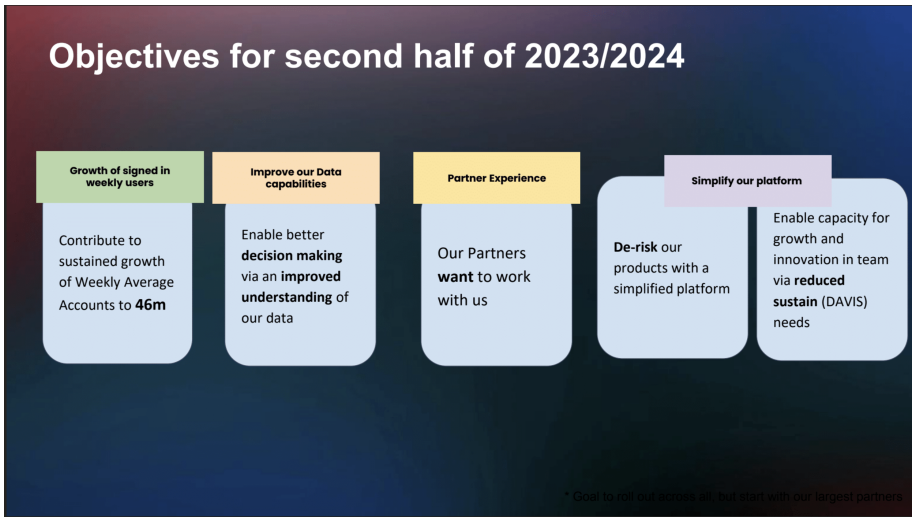


Figure 54: Image taken from a presentation given at a partnerships context setting event (BBC Partnerships, 2023).

9.2 Appendix B - Full schedules design including future external notification work

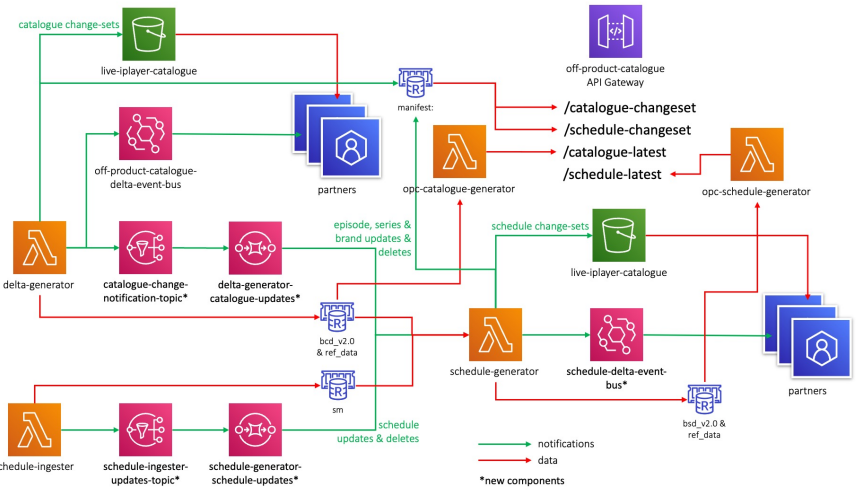


Figure 55: Full diagram of design for schedules pipeline, including future notifications to partners work (Lloyd, 2023).

9.3 Appendix C - Initial flow diagrams for how events would be processed

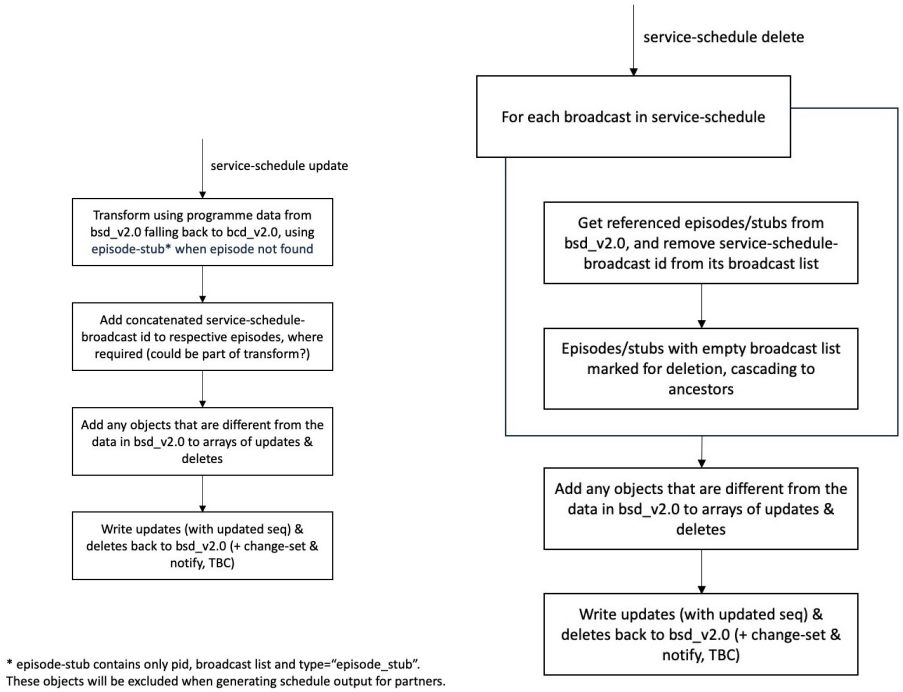


Figure 56: Flow diagrams for schedule events (Lloyd, 2023).

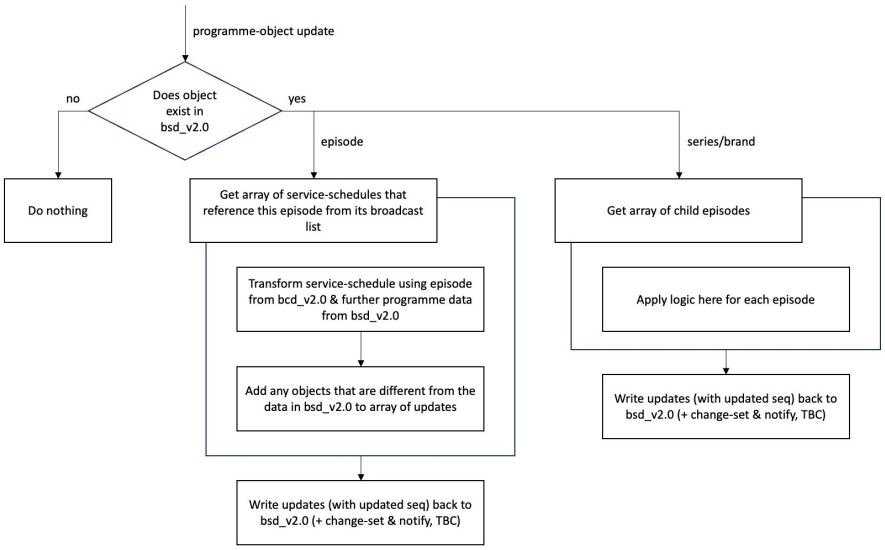


Figure 57: Flow diagram for catalogue/programme events (Lloyd, 2023).

9.4 Appendix D - Full ways of working diagram

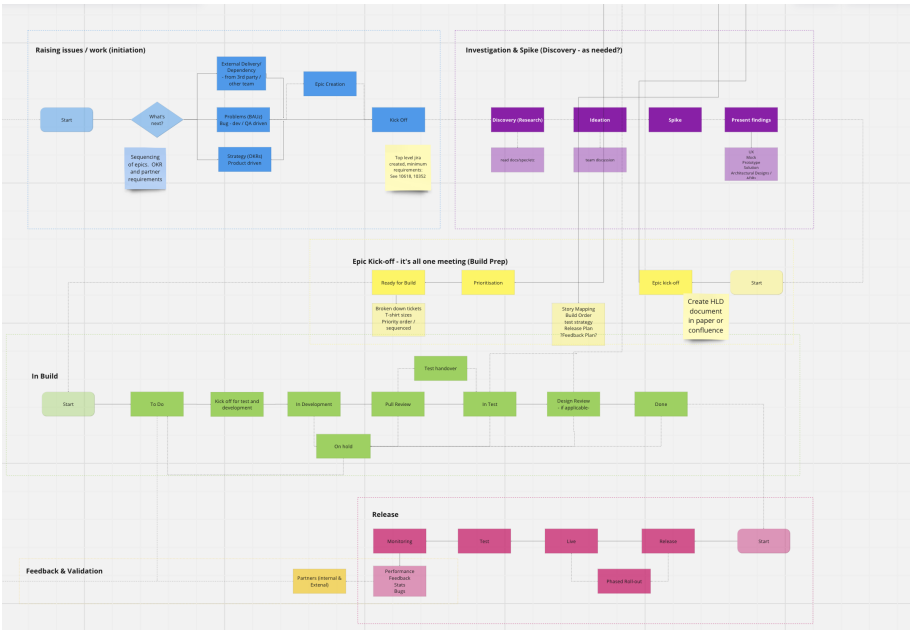


Figure 58: Full ways of working flow diagram used by SpaceChimp.

9.5 Appendix E - Full software spike document

Updating schedule-generator with deltas

Spike: Schedule-generator Deltas

- **Conducted by:** Fabrizio Colucci & Oli Bowker (with exec. consultants Ray Cheung & Karl Lloyd)
- **Backlog Work Item:** IPLAYERTVV1-14412

Goals

1. Validate proposed algorithms for dealing with service-schedule and catalogue updates and deletes flowing into the schedule-generator - no optimisations at this point
2. Outline what updates would be required to the existing cold-start mode to be compatible with this
3. Suggest what potential optimisations could be made, taking into consideration the current typical delta throughput from both sources

Method

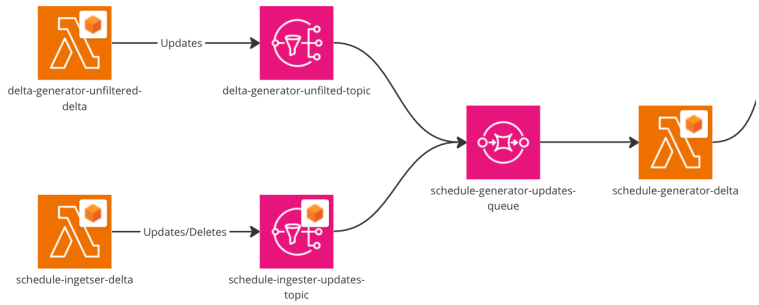
1. Build prototype to Int environment.
2. Use unit tests to cover list of scenarios in the above parent ticket.
3. Promote spike to Test environment to analyse capability of handling throughput of real data.
   a. Add a delta-generator outgoing SNS for notifications. Send only on Int & Test?
   b. Other infrastructure exists in schedule-generator spike branch - attach incoming SQS to existing schedule-ingester SNS, etc. Deploy this in parallel with current schedule-generator.
4. Analyse the complexity/maintainability of solution.

Test scenarios

See IPLAYERTVV1-13966 description for initial list of scenarios to cover.

Evidence

The repo and unit tests for algorithm and subsequent AWS architecture. The delta lambda can be found here and the logs for the lambda can be found here.



New architecture for the schedule-generator

Metrics have been setup in grafana to show throughput on TEST, under the `Schedule Generator (SPIKE work)` tab.

- Metric to track number of updates and deletes per type (episode/series/brand)
- Metric to track number of ignored updates and delete per type (episode/ancestor)
- Metric to track general lambda errors
- Metric tracking average run time, invocations and oldest message in queue.

Conclusions

Approach

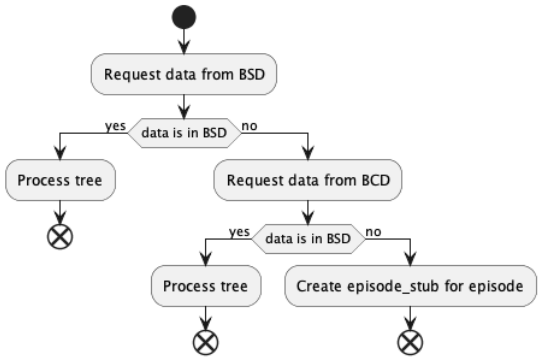
The final approach consisted of repositories requiring change.

- A spike repo was created that housed the new generator code. This code writes to the redis keyspace *{bsd_v2.0_spike}* and *{bsd_v2.0_ref_data_spike}*. Unit tests were written that covered scenarios specified in the intial breif.
- Updates to the delta generator were added to add an SNS topic on the unfiltered lambda, which posts **UPDATES** events only to the schedule generator SQS.

On schedule **UPDATE** all broadcasts are iterated through and respective data episode/series/brand data is persisted into the *{bsd_v2.0_spike}* keyspace by the below algorithm. With broadcaast lists of episodes being populated with an array of

`*sid|broadcast_pid*` identifiers. This list is used to know what schedules need updating when episode/series/brand updates come in.

- Edge case - broadcast_pid or episode is changed in an existing schedule. Remove old identifier from the old referenced episode, if that broadcast_list is empty then remove the episode.



Schedule update tree building logic.

On schedule **DELETE** all broadcast are iterated through, with associated `*sid|broadcast_pid*` identifiers being removed from the episodes broadcast_list. If this list ends up being empty, the episode is removed from *{bsd_v2.0_spike}* alongside the schedule.

On episode **UPDATE** get the new episode data from *{bcd_v2.0}* and then append the broadcast_list and new seq.

On series/brand **UPDATE** get the new series/brand from *{bcd_v2.0}* and then append a new seq number to it. Then determine the linked episodes in *{bsd_v2.0_spike}* and update the relevant schedules in their broadcast_lists.

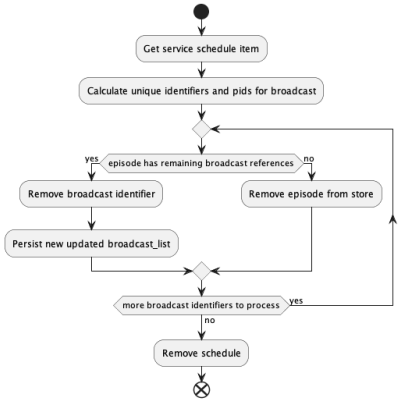
Catalogue **DELETES** are not explicitly handled.

Coldstart
Coldstart functionality was not fully implemented during the spike however below is a list of things that would need to be handled.

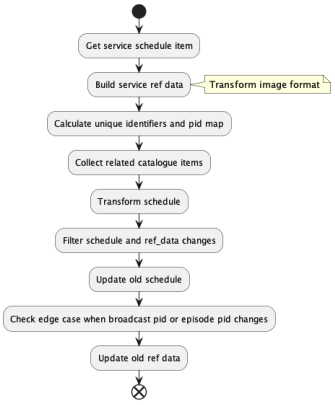
- Standard *'cut-off'* for the lamdba SQS mappings would need to be added for the SQS going into the schedule generator. This also includes perging the queue.
- Protection when the delta-generator coldstarts. Implement some form of threshold of amont of updates that can be sent to the schedule-generator at once. If this threshold is breached, coldstart the schedule generator as we do in the aggricat.
- Purge all catalogue objects from *{bsd_v2.0}* and compile episodes/series/brands.

Garbage Collector

The current system only handles the deletion of episodes and schedules from the *{bsd_v2.0_spike}* keyspace. It was determined a single job ran daily to clean up the series and brands would be better as all data from the *{bsd_v2.0}* keyspace would have to be read and parsed for every delete. Considering deletes only happen once a day, we may as well tidy up once a day.



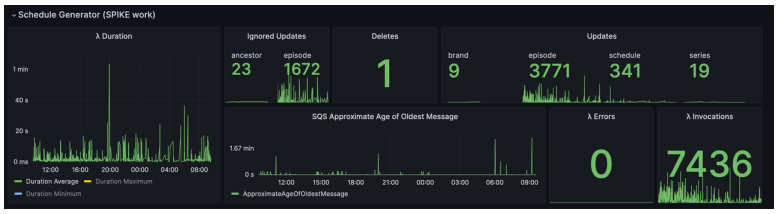
Activity for schedule delete.



Activity for schedule update.

Monitoring

Below is moniroing stats for a 24 hour period. It's important to note the ignored updates, actual updates and deletes will all be higher when we do this for real. This is due to the spike not coldstarting, therefore there are less schedules in which in the store. However this does nicely show a *maximum* amount of events we're likely to recieve.



Pros	Cons
Schedules can somewhat rely on it's own data store after initial setup of data is done.	**Large duplication of redis data across {bsd_v2.0} and {bcd_v2.0}.**
New code and infrastructure is all things that we have done before, nothing new to learn.	**Hard to parallelise if we choose to.**
Changesets should be easier to implement using a solution like this.	**Catalogue and Schedule pipelines are not separate, (however this is already how it works on LIVE).**

Potential Improvements

- Schedule-generator
  - Parrelise schedule updates triggered by episode/series/brand changes. An episode change then triggers updates for any schedule that it refers to, brand and series also do this but for the entire tree. An episode/series/brand present in BBC One can therefore will spin off a schedule change for all regional and hd variants. If a brand like eastenders was to change for whatever reason, this would result in 100s of schedule changes. This is a must.
  - Remove all catalogue deletes from schedule generator, and tidy up anything with an empty broadcast_list, and it's parents in the garbage collector. This would simplify the current schedule-generator code and make garbage collection easier.
  - Implement episode→broadcast mappings in DynamoDB instead of having broadcast_list in episodes. This would remove the need for redis duplication and would allow us to parallelise using optimistic locking in the future if we choose/need to.

        - This could be done at the ingester, once more simplifying the schedule-generator even more.
- Delta-generator
    - More filtering could be applied as we only care if titling, synopses, subtitles, (images?) or warning_text has been updated. If a dynamoDB mapping approach is taken, then a lookup could be done and only relevant pids would be sent on top of this (maybe over engineering).
    - Make code more explicit that it wants to publish to SNS, don't rely on their not being a manifest key and the version not being v1.1.

Next Steps

If we are happy to continue with the above approach, Oli and Fab will create/slice tickets and a kick off can be planned.

9.6 Appendix F - Architectural decision for single schedules store

Architecture Decision Record 022: Single source for schedule data

Context

Schedule documents rely on catalogue data for titling, programme descriptions, subtitling and viewer discretion data. When requesting schedules via the API Gateway the partner also receives episode, series and brand data associated with the schedules requested as part of the response. This data can be retrieved from the catalogue redis store but the schedules pipeline is then fully reliant on the catalogue pipeline. This reliance of the catalogue pipeline (*bcd_v2.0:*) means that the schedule API gateway is not independent therefore it was requested to make catalogue data more integrated within the schedule redis stores. This ADR describes the decision made to ensure data independence.

Decision

Copy over all catalogue data currently relating to schedules into the keyspace *bsd_v2.0:* and *bsd_v2.0_ref_data:* respectively on notification events or coldstart. This ensures the API Gateway can retrieve all of its information from one source and keep it's integrity and self-consistency. This data should also be removed when no longer referenced by any schedules.

Status

Accepted

Consequences

- Overhead of copying over data existing data.
- Added maintenance and code complexity.
- Potential synchronisation issue when copying/deleting data across redis stores.

9.7 Appendix G - !TODO! List of e2e test scenarios

Updating schedule-generator with deltas

Spike: Schedule-generator Deltas

- **Conducted by:** Fabrizio Colucci & Oli Bowker (with exec. consultants Ray Cheung & Karl Lloyd)
- **Backlog Work Item:** IPLAYERTVV1-14412

Goals

1. Validate proposed algorithms for dealing with service-schedule and catalogue updates and deletes flowing into the schedule-generator - no optimisations at this point
2. Outline what updates would be required to the existing cold-start mode to be compatible with this
3. Suggest what potential optimisations could be made, taking into consideration the current typical delta throughput from both sources

Method

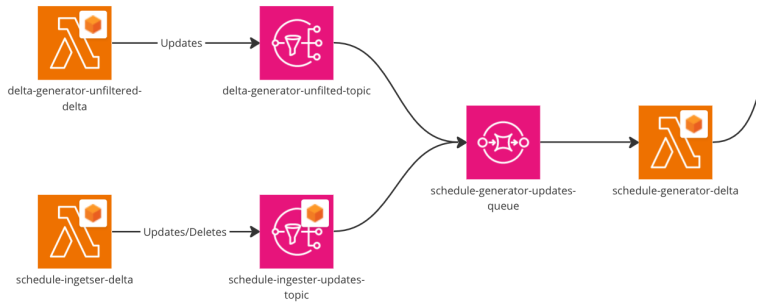
1. Build prototype to Int environment.
2. Use unit tests to cover list of scenarios in the above parent ticket.
3. Promote spike to Test environment to analyse capability of handling throughput of real data.
   a. Add a delta-generator outgoing SNS for notifications. Send only on Int & Test?
   b. Other infrastructure exists in schedule-generator spike branch - attach incoming SQS to existing schedule-ingester SNS, etc. Deploy this in parallel with current schedule-generator.
4. Analyse the complexity/maintainability of solution.

Test scenarios

See IPLAYERTVV1-13966 description for initial list of scenarios to cover.

Evidence

The repo and unit tests for algorithm and subsequent AWS architecture. The delta lambda can be found here and the logs for the lambda can be found here.



New architecture for the schedule-generator

Metrics have been setup in grafana to show throughput on TEST, under the `Schedule Generator (SPIKE work)` tab.

- Metric to track number of updates and deletes per type (episode/series/brand)
- Metric to track number of ignored updates and delete per type (episode/ancestor)
- Metric to track general lambda errors
- Metric tracking average run time, invocations and oldest message in queue.

Conclusions

Approach

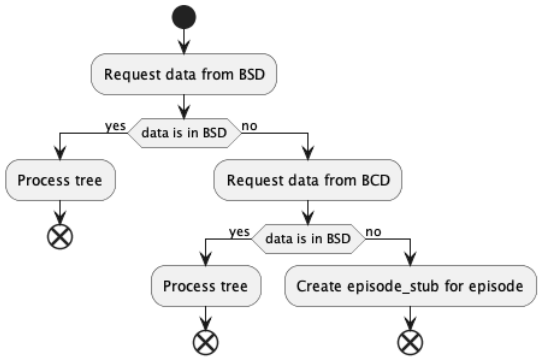
The final approach consisted of repositories requiring change.

- A spike repo was created that housed the new generator code. This code writes to the redis keyspace *{bsd_v2.0_spike}* and *{bsd_v2.0_ref_data_spike}*. Unit tests were written that covered scenarios specified in the intial breif.
- Updates to the delta generator were added to add an SNS topic on the unfiltered lambda, which posts **UPDATES** events only to the schedule generator SQS.

On schedule **UPDATE** all broadcasts are iterated through and respective data episode/series/brand data is persisted into the *{bsd_v2.0_spike}* keyspace by the below algorithm. With broadcaast lists of episodes being populated with an array of

`*sid|broadcast_pid*` identifiers. This list is used to know what schedules need updating when episode/series/brand updates come in.

- Edge case - broadcast_pid or episode is changed in an existing schedule. Remove old identifier from the old referenced episode, if that broadcast_list is empty then remove the episode.



Schedule update tree building logic.

On schedule **DELETE** all broadcast are iterated through, with associated `*sid|broadcast_pid*` identifiers being removed from the episodes broadcast_list. If this list ends up being empty, the episode is removed from *{bsd_v2.0_spike}* alongside the schedule.

On episode **UPDATE** get the new episode data from *{bcd_v2.0}* and then append the broadcast_list and new seq.

On series/brand **UPDATE** get the new series/brand from *{bcd_v2.0}* and then append a new seq number to it. Then determine the linked episodes in *{bsd_v2.0_spike}* and update the relevant schedules in their broadcast_lists.

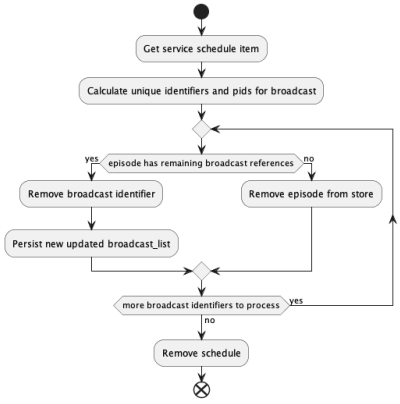
Catalogue **DELETES** are not explicitly handled.

Coldstart
Coldstart functionality was not fully implemented during the spike however below is a list of things that would need to be handled.

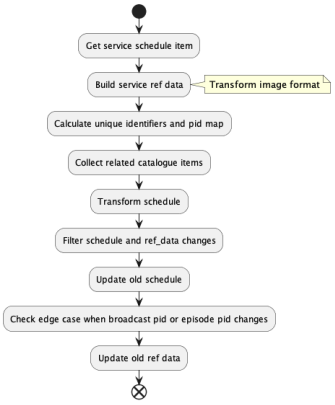
- Standard *'cut-off'* for the lamdba SQS mappings would need to be added for the SQS going into the schedule generator. This also includes perging the queue.
- Protection when the delta-generator coldstarts. Implement some form of threshold of amont of updates that can be sent to the schedule-generator at once. If this threshold is breached, coldstart the schedule generator as we do in the aggricat.
- Purge all catalogue objects from *{bsd_v2.0}* and compile episodes/series/brands.

Garbage Collector

The current system only handles the deletion of episodes and schedules from the *{bsd_v2.0_spike}* keyspace. It was determined a single job ran daily to clean up the series and brands would be better as all data from the *{bsd_v2.0}* keyspace would have to be read and parsed for every delete. Considering deletes only happen once a day, we may as well tidy up once a day.



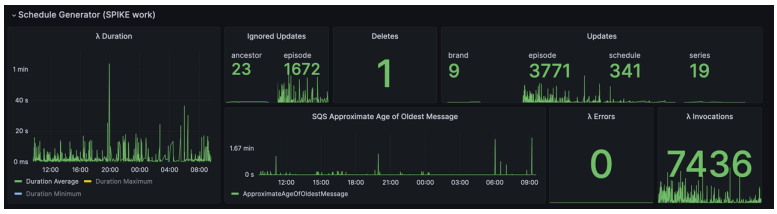
Activity for schedule delete.



Activity for schedule update.

Monitoring

Below is moniroing stats for a 24 hour period. It's important to note the ignored updates, actual updates and deletes will all be higher when we do this for real. This is due to the spike not coldstarting, therefore there are less schedules in which in the store. However this does nicely show a *maximum* amount of events we're likely to recieve.



Pros	Cons
Schedules can somewhat rely on it's own data store after initial setup of data is done.	**Large duplication of redis data across {bsd_v2.0} and {bcd_v2.0}.**
New code and infrastructure is all things that we have done before, nothing new to learn.	**Hard to parallelise if we choose to.**
Changesets should be easier to implement using a solution like this.	**Catalogue and Schedule pipelines are not separate, (however this is already how it works on LIVE).**

Potential Improvements

- Schedule-generator
  - Parrelise schedule updates triggered by episode/series/brand changes. An episode change then triggers updates for any schedule that it refers to, brand and series also do this but for the entire tree. An episode/series/brand present in BBC One can therefore will spin off a schedule change for all regional and hd variants. If a brand like eastenders was to change for whatever reason, this would result in 100s of schedule changes. This is a must.
  - Remove all catalogue deletes from schedule generator, and tidy up anything with an empty broadcast_list, and it's parents in the garbage collector. This would simplify the current schedule-generator code and make garbage collection easier.
  - Implement episode→broadcast mappings in DynamoDB instead of having broadcast_list in episodes. This would remove the need for redis duplication and would allow us to parallelise using optimistic locking in the future if we choose/need to.

        - This could be done at the ingester, once more simplifying the schedule-generator even more.
- Delta-generator
    - More filtering could be applied as we only care if titling, synopses, subtitles, (images?) or warning_text has been updated. If a dynamoDB mapping approach is taken, then a lookup could be done and only relevant pids would be sent on top of this (maybe over engineering).
    - Make code more explicit that it wants to publish to SNS, don't rely on their not being a manifest key and the version not being v1.1.

Next Steps

If we are happy to continue with the above approach, Oli and Fab will create/slice tickets and a kick off can be planned.

9.8 Appendix !TODO! - Competency Mappings and Links

Competency	Description	Page
B1	Identify, document, review, and design complex IT-enabled business processes that define a set of activities that will accomplish specific organisational goals and that provide a systematic approach to im- proving those processes.	Page X
P3	Professionally present digital-and-technology-solution-specialism plans and solutions in a well- structured business report.	Page X
P4	Demonstrate self-direction and originality in solving problems, and act autonomously in planning and implementing digital-and-technology- solution-specialist tasks at a professional level.	Page X
P5	Be competent at negotiating and closing techniques in a range of interactions and engagements, both with senior internal stakeholders and external stakeholders.	Page X
SE-S01	Architect, build, and support leading-edge concurrent-software plat- forms that are performant to industry standards and that deliver responsive solutions with good test coverage.	Page X
SE-S02	Drive the technology-decision-making and development process for projects of varying scales, considering current technologies including DevOps and Cloud Computing, and evaluate different technology- design and implementation options, making reasoned proposals and recommendations.	Page X
SE-S03	Develop and deliver distributed or semi-complex software solutions that are scalable, and that deliver innovative user experiences and journeys that encompass cross-functional teams, platforms, and technologies.	Page X
SE-S04	Update current software products, improving their efficiency and functionality, and build new features to product specifications.	Page X
SE-S05	Accomplish planned software-development tasks that deliver the expected features within specified time constraints, security, and quality requirements.	Page X
SE-S06	Be accountable for the quality of deliverables from one or more software-development teams (source code quality, automated testing, design quality, documentation, etc.), and following company-standard processes (code reviews, unit testing, source code management, etc.)	Page X
SE-K02	The various inputs, statements of requirements, security considerations and constraints that guide solution architecture and the development of logical and physical systems' designs.	Page X
SE-K03	The methodologies designed to help create approaches for organizing the software-engineering process, the activities that need to be undertaken at different stages in the life-cycle, and techniques for managing risks in delivering software solutions.	Page X

SE-K04	The approaches used to modularise the internal structure of an application, and to describe the structure and behaviour of applications used in a business, with a focus on how they interact with each other and with business users.	Page X
SE-K05	How to design, develop, and deploy software solutions that are secure and effective in delivering the requirements of stakeholders, and the factors that affect the design of a successful code.	Page X
SE-K06	The range of metrics which might be used to evaluate a delivered software product.	Page X