

Bringing DataOps to Power BI

11 Jul 2021 3 mins read

I need better equipment I tell myself while trying to jog off the extra pounds I gained in 2020. The earphones I am wearing slowly loosen from my ears with each step and I can feel each contour of the blacktop with my worn-out sneakers. I manage to finagle the earphones just right to listen to a new podcast [Building the Backend](#) by Travis Lawrence. He is interviewing Chris Bergh from Data Kitchen and a principal author of DataOps. I had heard the term before, DataOps, and read the manifesto, read the DataOps Cookbook, and I had researched how DataOps is the fusion of DevOps, Agile, and Lean Manufacturing. The stories Chris and his colleagues share about clients calling about bad data in reports, ETL pipelines failing in production, and clients who thought new features could be delivered quicker were all experiences I could relate with. I had been a convert for a little over a year and wanted to hear more.

As I slogged up a hill trying to convince myself to keep going, Travis asked Chris the question:

“Where do you see data architectures hitting in the next two to five years?”

To paraphrase Chris sees two camps emerging:

1. The low-code side, tools that generate code but just in the background using fancy interfaces; furthermore, these are geared towards data analysts and folks without a software development background.
2. The high-code side, tools that help build data lakes, data warehouses and they are tailored for those with software development backgrounds.

And Chris sees that the challenge for organizations will be how they define the processes that adhere to DataOps principles in order for the two camps to co-exist and bring value to the organization.

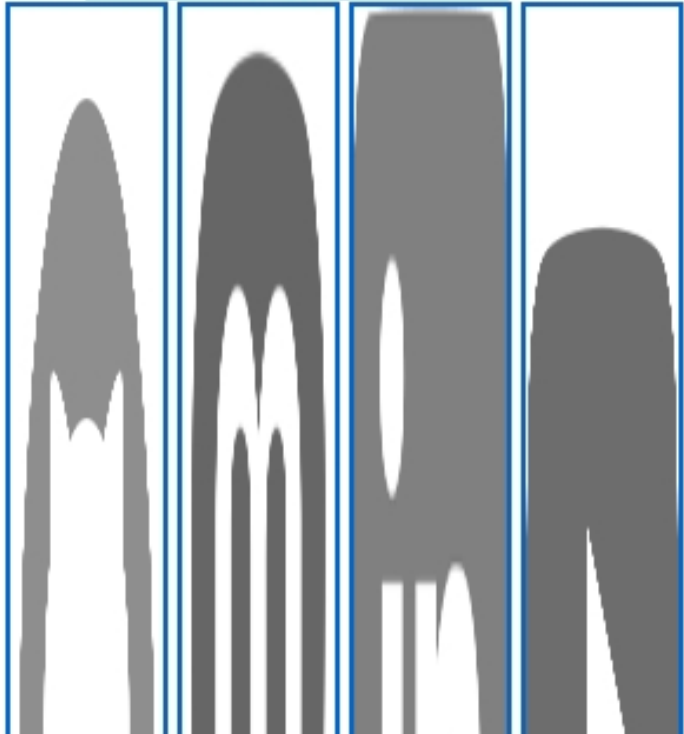
I understood exactly what he meant. For well over a year I have been working in both camps. On the high-code side, I was working SQL, C#, and JavaScript working with teams to build solutions with unit tests, implementing continuous integration, and having tools like Application Insights and StackTraceJS to proactively address issues with clients.

On the other side though, there was Power BI. Please don’t get me wrong, I like the tool and its capabilities are profound and definitely a reason I have stable employment. But across version control, testing, continuous integration, to monitoring, applying DataOps principles to Power BI is a challenge *I need better equipment*.

It has taken me the better part of a year, but I think I have put together some concepts and accelerators to bring DataOps to Power BI, and I would like to share them on this blog with the tech community. Along the way I will touch on a DataOps principle, the challenges adhering to that principle with Power BI, and my thoughts on overcoming those challenges.

My hope is that you find these concepts and accelerators helpful, but I would like your feedback. My thoughts come from my experiences, my work with clients and their environments. Please let me know what you think on LinkedIn or Twitter. Maybe there will be enough discussions that eventually we improve upon these concepts and accelerators, or they make their way into the Power BI product. If DataOps principles become more inherent to Power BI’s feature set and this blog series becomes obsolete, all the better.

Next [Part 2: Bringing DataOps to...](#)



Part 2: Bringing DataOps to Power BI

20 Jul 2021 6 mins read

DataOps Principle #16 - Monitor for Quality and Performance

Now before I talk about this principle, I would like to provide some context. Most of the time my work is done without the privilege of being a Power BI administrator. The most privilege I get is administrator rights to a Power BI workspace. I have worked in some large enterprise environments and separation of privileges is paramount; thus, our teams do not have access to certain features a Power BI administrator may have that would better serve this principle.

“Our goal is to have performance, security and quality measures that are monitored continuously to detect unexpected variation and generate operational statistics.”

From my point of view the two words “monitored continuously” stand out in this principle and when it comes to Power BI the content that must be frequently checked for issues is the refreshing of the data. Power BI provides automated refreshes, incremental refreshes, refreshes with on-premise data (via gateways), and more. Yet, given our privileges my colleagues and I have been met by some challenges with monitoring for quality and performance within Power BI.

The Challenges

- 1) Email Alerts and Automated Scheduled Refreshes** “What happens when a refresh fails? An email is sent to a predefined group email or individual. If your inbox looks anything like mine in a given day, the number of alerts are overwhelming and there is a good chance you may miss a failed refresh. I have also fallen prey to the Focused Inbox feature in Outlook that, by default, label those alert notices as “Other” thereby relegating the important messages into obscurity and allowing latent data to persist in the Power BI reports. Best I can gather is I am not training the Machine Learning model well enough to make my Focused Inbox better, but do I need to? And should I expect my colleagues to do the same?
- 2) Gateways and Deactivating Consecutive Failed Refreshes** - As of July 2021, Power BI deactivates the refresh scheduled after four consecutive failures, or the service encounters invalid or expired credentials. In the environments I work within, I am often not the gateway administrator, and I am not the administrator of many of the source systems. If our team has scheduled an automated refresh to run hourly and a database goes down or a gateway has a network issue while we are asleep, there is a good chance I wake up with emails from a client asking why the data has not been refreshed since yesterday. Furthermore, my team has to spend time going through the datasets to see which ones may have been disabled and re-enable them. That is overhead that is not necessarily delivering value.
- 3) Refresh Time Analytics** “Power BI’s Refresh History is helpful for diagnosing refresh failures, but what about the time it takes to refresh? It tells me the start and end times, but I have to figure out how many minutes transpired AND I only get a limited history so I cannot really tell if, for example, 10 minutes is acceptable for refreshing this dataset. And even if I may know that is not acceptable, how does the person covering for me when I want to take a vacation know that is not acceptable? What context do they have to make an informed decision? Finally, and I feel like I am piling on now, why do I have to close and open the Refresh history modal to get the latest status of a refresh in progress? I find our team, clicking that the “Close” button and “Refresh History” link incessantly.

☒ The Refresh History modal does not update the GUI in real-time.

Writing these challenges is a little cathartic for me, but when it comes down to adhering to the principle of “Monitor for Quality and Performance” within Power BI we need to be able to answer the key question:

How do we effectively operationalize the monitoring of refreshes?

Best Practices

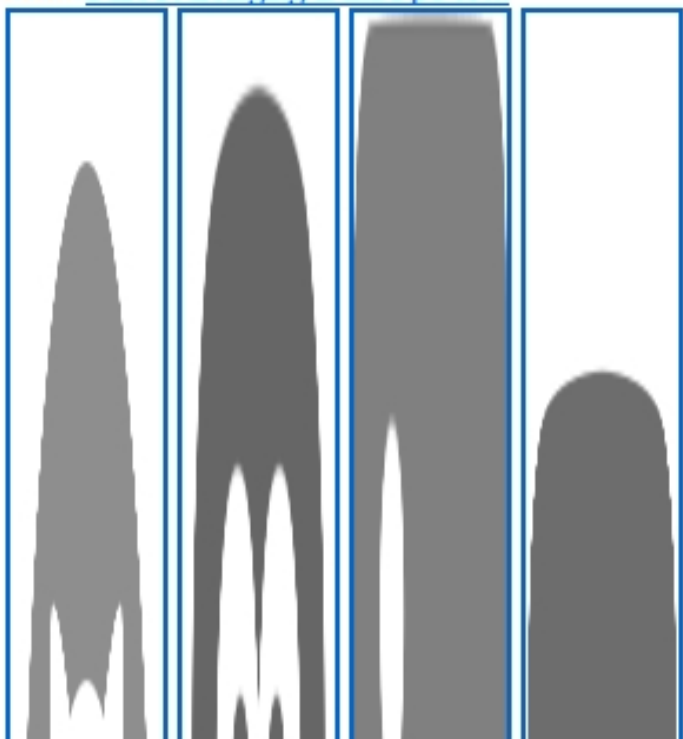
- The answer to question 1 is, first and foremost DO NOT rely on emails. I am sure this is shocking to Site Reliability Engineers and DevOps Engineers reading this, but chances are the folks charged with keeping the Power BI reports in good shape will have alert fatigue with email. Instead, designate a person on your team to monitor for failed refreshes. If you are following DataOps principle #5, Daily Interactions, and your refresh schedules can accommodate it, have the designee check right before the daily standup and have it as a 16th minute item (Heather and Greg, you know who you are, that was a good idea on our last project). Moreover, give that designee the right equipment with a Power BI Report to keep an eye on Power BI. Below is a simple report that uses Power BI APIs and it provides the following:
- 1) A simplified card that indicates if any failed refreshes occurred within the last 72 hours on your workspace. If it is green and 0, we are good. If it is red with a number greater than zero, click on the card and you will see what datasets have issues.
 - 2) A straightforward table that indicates if any datasets listed have had their automate refreshes disabled and, thus, need to be re-enabled. No more manually checking each dataset.
 - 3) A listing of median refresh rates for each dataset. A Power BI novice can set conditional formatting on this visual to turn red if the median time surpasses a concerning threshold.
 - 4) Finally, and most importantly, the report has a “Data As Of” timestamp. You are going to have this report on its own automated refresh schedule. Train those monitoring the refreshes with the report to make sure that timestamp is recent!

☒ Sample Report for Monitoring Quality and Performance. I have noted the aforementioned features listed with numbered circles.

I have shared this report template [on GitHub at this link](#). This is a version that works with a service account assigned a Premium Per User license (\$240/year and well worth the investment). You will need to update some parameters within Power Query to connect to your workspace correctly and I geared it towards the U.S. Commercial version of Power BI. I added a readme to this folder in the repo that explains about the inner workings and configuration. If you’re interested in a version that works with Power BI Premium (using an app service principal) let me know.

In the meantime, I’d love to hear your thoughts? Have you had similar experiences? Is this helpful? Let me know on LinkedIn or Twitter, please.

Previous [Bringing DataOps to Power BI](#)
Next [Part 3: Bringing DataOps to...](#)

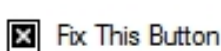


Part 3: Bringing DataOps to Power BI

27 Jul 2021 5 mins read

DataOps Principle #11 – Make it reproducible

I am panicked – it has been a long week at work and a small group of Power BI reports are showing the dreaded “Fix this” error. First piece of advice, the “Fix this” button is not an “Easy button” and pressing it will blow away the offending issue and obliterate all the formatting you have painstakingly put in place. Click on the “See details” first and see if you can fix the measure or calculated column (often the issue) and only press the “Fix this” button if you’re ready to take on the rework.

 Fix This Button *The dreaded Fix this button.*

So, I ask my project team what the “See details” is indicating. They say that a core measure in the Power BI dataset has gone missing and several measures in our reports rely on it, ergo the error message. What is worse is that we don’t know when it went missing! So, each member goes hunting where most Power BI reports reside – the Downloads folder. We scour the numerous copies of Power BI files suffixed by a parenthesis and number (ex. File (2).pbix) and try to find the lost measure.

 Picture of Downloads Folder *Does your Download folder keep your versions of Power BI?*

After 30 minutes we find multiple versions – now which one is right? After another 30 minutes of trial-and-error, we find the right version and fix the model. Crisis averted, but to me this was not acceptable and quite embarrassing.

DataOps Principle #11 - Make it reproducible: Reproducible results are required and therefore we version everything: data, low-level hardware and software configurations, and the code and configuration specific to each tool in the toolchain.

I share this story because I have heard similar stories from other Power BI teams, so if you have had a similar experience, you’re not alone. For me, the Power BI product poses a serious challenge to the DataOps principle “Make it reproducible” because versioning our Power BI faces the following challenges:

The Challenges

1) **The Power BI format** – A Power BI file is a zip file of many files, so unlike other programming languages (ex. C#, Java) where you can use version control to merge changes within a file, Power BI doesn’t have that capability. And if you use third party tools to edit these files (like the lovely Tabular Editor) you operate under your own risk and should you corrupt the pbix file, Microsoft Support will not help you. This is a common pattern with low-code solutions, so Power BI is not alone here, but it is vexing nonetheless.

 Pbx File Unizpped with Folders *A pbix file under the hood*

2) **The Coupling of Code, Data, and Visualizations** – The custom code we develop in Power BI is often DAX (Calculated Columns, Calculated Tables, and Measures) or M language (Power Query) and it is tightly coupled with the underlying data AND the visualizations of that data. It is convenient as a low-code solution, but DataOps emphasizes decoupling each component and versioning each.

Best Practices

First and foremost, my advice for best practices is not a panacea. It will however mitigate the risk of version issues and make it much easier to manage a team working with Power BI.

1) **Get Version Control** – This shouldn’t shock you, but if you don’t use Git to store your Power BI files you really need to use it now. Tools like Azure DevOps (free for up to 5 users), Bitbucket, and GitHub mean you shouldn’t have an excuse to version your Power BI code. If you’re unfamiliar with Git, [GitHub Desktop](#) makes it super simple to commit and push your Power BI file changes to a repo. It also forces you and your team to provide a description of the changes you make and allows you to recover prior versions should the latest version of your Power BI file have an issue.

2) **Separate Reports From the Model** – I’ve heard many [Microsoft MVPs and employees talk about this](#), but I’m repeating it. Following this advice is quite helpful because:

- Many reports can look at one version of the data which eliminates the overhead of fixing the same measure several times; you just make the change once.
- You can stub out your model by manually entering in data in Power Query to create tables, hardcoding your DAX measures, and then publishing a model to DEV. Your other team members won’t have to wait for you to finish working on the model and they can be productive building the visualizations and delivering products faster. In fact, you can take it a step further and have your team members work different parts of the report in parallel on separate Power BI files and then combine them through Power BI’s copy and paste feature. I’ve done this on numerous projects now and it has really sped up delivery time.

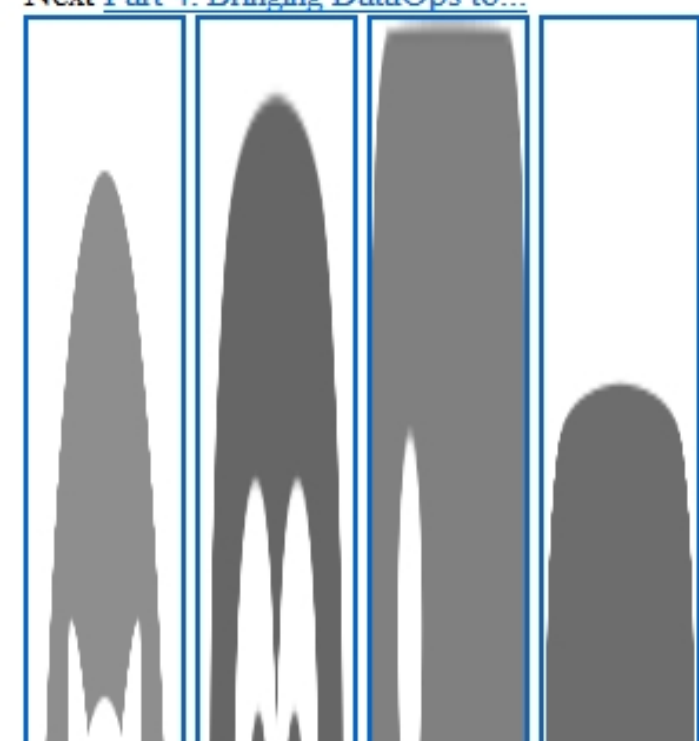
 Easily copy Power BI visuals when using a shared dataset. *By separating the reports from the model you can work separately on Power BI reports and combine work using copy and paste.*

3) **Institute Workspace Governance** – Do not intermingle your development Power BI files with the Power BI files that your clients/customers use. You should at a minimum have two workspaces, one for development (DEV) and one for production (PROD). For larger projects, you should have a workspace for clients/customers to test (TEST) before moving to production. If you are unfamiliar with the concept please read [this wiki article](#).

4) **Workspace Permissions** – For larger teams limit those who have Member or Admin permissions to your TEST and PROD environments. Those with Member or Admin permissions should only publish Power BI files if it is in source control. If you have Premium, try [Deployment Pipelines](#). And if you wait a few more weeks, I will show you how to automate the testing and deployment of Power BI files with Azure DevOps and Power Shell (yep, shameless teaser).

Previous [Part 2: Bringing DataOps to...](#)

Next [Part 4: Bringing DataOps to...](#)



Part 4: Bringing DataOps to Power BI

03 Aug 2021 5 mins read

DataOps Principle #15: Quality is Paramount

Yep, I did it to myself! I thought I had fixed an issue in the data model, but I ended up creating a new one. A client had called about a calculation issue, and I had opened Power Query to investigate. I was looking at several steps in Power Query and to help with my investigation I applied a filter step to Table X. As I navigate to Tables Y and Z, I find the issue, apply a fix to Table Z and close Power Query. So, I forgot to remove the filter on Table X and I published the dataset, only to create another issue that I learned about hours later. Thankfully we had instituted [best practices in version control](#) so the dataset didn't go to production with the new error, but I was frustrated that the mistake was so easy to make. Also, I found it vexing that there was no safety net for either me or my colleagues to keep us from falling flat on our faces with these kind of easy mistakes. Our team needed a safety net.

 Applying a Filter in Power Query *Have you applied a filter on a table in Power Query and forget to remove it later?*

DataOps Principle #15 - Analytic pipelines should be built with a foundation capable of automated detection of abnormalities and security issues in code, configuration, and data, and should provide continuous feedback to operators for error avoidance.

The biggest challenge to applying this principle within Power BI is the lack of concrete testing frameworks. Yes, there are great third-party tools (like Dax Studio) to help conduct tests, but not necessarily a great framework to instill good testing of Power BI. How do we consistently empower Power BI developers to build sound models, build tests easily, and establish a basis of tests to protect the development team from regression errors.

Best Practices

I have learned if you follow these best practices, you won't eliminate all errors, but you sure will reduce them.

1) Have a Test Dataset – In order to test effectively you need to have a known underlying dataset to validate your Power BI report. For example, if you are importing a SQL database you don't adjust the query to be a window of time in the past or clone the data. The goal is to have a static set of data to work with, so the only variables that would change during a test is the code you or your team has changed in Power BI.

2) Test The Tables – As of August 2021 you can't test each line of M code in Power Query that is generated when you build your tables. However, if you follow the prior best practice you can test to make sure the composition of the tables created by your code are well-formed. For example, you could test for:

- Number of rows in a fact table.
- Number of rows in a dimension.
- The presence of a value in a column that shouldn't be there.
- The existing of blank columns.
- The values in a custom column are correct.

So, in the case where you apply a filter to troubleshoot your Power BI file and forget to remove it, a properly defined test case would indicate the issue.

3) Test The DAX – If you follow the first best practice you should be able to test that a DAX measure or calculated column has the appropriate result. For example, let's say you have a DAX measure $\text{IF}(\text{SUM}(\text{TableX}[\text{ColumnY}]) < 0, \text{Condition 1}, \text{Condition 2})$. To test properly you should create conditions to test when:

- The summation is > 0
- The summation is $= 0$
- The summation is < 0
- The summation is blank

These types of tests are not just for numeric-based measures/columns. You can also test for potential issues with time zones, when source columns are blank, etc.

4) Have a Test Schema – All tests should result in an output that is consistent so that the team can understand the results and find failed tests quickly. Fortunately, with DAX, we can create that schema. Figure 2 is an example of the output of a set of test cases and follows a simple format:

- Test Name- Description of the test prefixed by MEASURE, CALCULATED COLUMN, or TABLE to indicate what part of Power BI model is being tested.
- Expected Value - What the test should result in. This should be a hardcoded value or function evaluated to a Boolean.
- Actual Value - The result of the test under the current dataset.
- Test Passed - True if the expected value matches the actual value. Otherwise, the result is false.

 Example of Test Cases *An example of test cases and their output.*

5) Maintain Your Tests – To paraphrase Edsger Dijkstra, a famous computer scientist, testing will show the presence of flaws, not the absence of flaws. Therefore, as you encounter more issues (and you will) make sure to update the test dataset and the test cases to check for the presence of the issues. The goal is to never encounter the same issue twice in production!

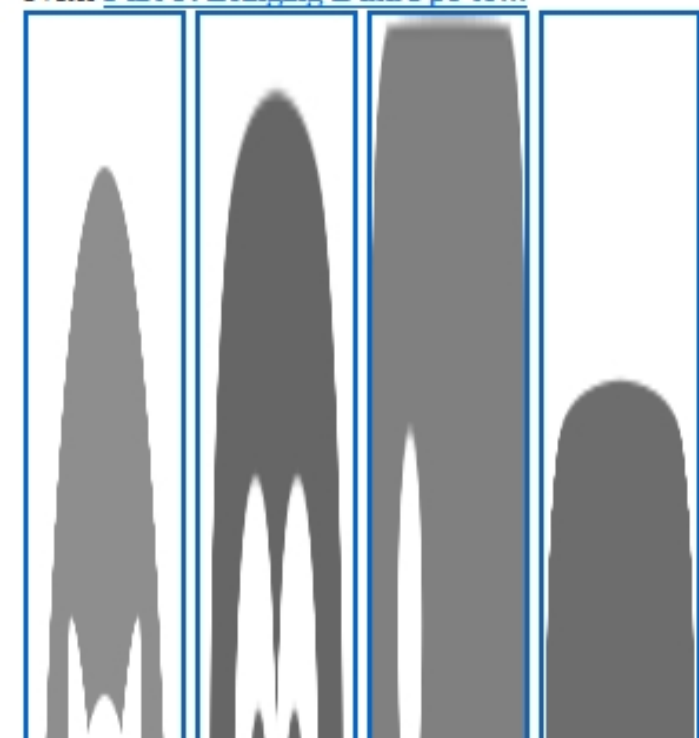
6) Version The Tests – Just as I recommended in [Part 3](#) that you should use version control with Power BI files, you should also include the tests when you check-in/commit your Power BI file. That way your colleagues can run the same tests to check for errors.

If you're interested, I've shared [an example of the tests along with a Power BI model on GitHub](#). I've also included instructions on how to run the tests using DAX Studio (a great product!).

In my next post for the series, I will combine the [best practices in version control](#) and the sample test cases and describe how to automate the testing and deployment of a Power BI file with Azure DevOps and PowerShell.

Previous [Part 3: Bringing DataOps to...](#)

Next [Part 5: Bringing DataOps to...](#)



Part 5: Bringing DataOps to Power BI

16 Aug 2021 5 mins read

DataOps Principle #10: Orchestrate

Let's Orchestrate! If you have read Part 3 and Part 4 of this series, you've probably noticed that I was building towards something. By combining the best practices in version control and the best practices in testing, you can start to automate the building and testing of Power BI reports/datasets. That means your team can avoid putting errors in production, test if refreshes work, and manage how a report gets into a Power BI Workspace. Everything I discuss in this entry will be made [available on GitHub](#). I followed the "Make it Reproducible" principle so I scripted out much of the steps using PowerShell and Azure CLI. My hope is you can use this as a template for your projects.

The beginning-to-end orchestration of data, tools, code, environments, and the analytic teams work is a key driver of analytic success.

If I was to offer a theme around this principle it would be: Continuous Integration (CI). Keep in mind that DataOps includes DevOps principles, so my definition of CI within Power BI is the practice of automating the integration of code changes from multiple contributors into a Power BI workspace. CI typically consists of 4 workflows, and to provide better context, the following is my commentary on how we map them to Power BI:

- 1) **Run tests locally** – This workflow is on the "honor system" where a developer follows best practices from Part 4 of the series.
- 2) **Compile code in CI** – For many development environments this is implemented using a build agent, a virtual machine/container that compiles the code and fails if the build fails. The idea here is the build agent standardizes the build to ensure the build succeeds and the later steps can be completed. With Power BI, I would consider the build agent a **Build** workspace, a temporary workspace in the Power BI service where we can test refreshes before it's placed in a Development workspace shared by everyone on the development team. If you work in the high-code side this may feel unusual, but on the low-code side this is becoming common place. The other parts of the Power Platform (PowerApps and Power Automate) have [introduced environments](#) and [Zachary Johnson](#) has written [about using environments in PowerApps for CI](#).
- 3) **Run tests in CI** – Using Azure DevOps and PowerShell we take the recently refreshed Power BI report in the Build workspace and run through all the tests (see [Part 4](#)) using XMLA. It's critical we use at least Power BI Premium Per User in order to have access to XMLA.
- 4) **Deploy an artifact from CI** – If the compilation (a.k.a. refresh) and tests complete, the Power BI dataset is ready to move to the Development workspace.

Putting Theory to Practice

Alright enough definitions and theory, you probably want to know how it all works. Here is the workflow:

☒ Continuous Integration Flow *The Steps for a CI pipeline*

Step 1 – Sounds simple, but let me take the opportunity to define the folder structure for the code

☒ Example folder structure for a Power BI project

The folder structure for a Power BI project

Let's look at the main folders:

- **PBI** – Within its own subfolder, this is where you would store a .pbix file and its tests (.dax and .msdax). In the image above, SampleModel.pbix file and 3 test files would be stored in the SampleModel subfolder.
- **PipelineScripts** – These files inform how the orchestration happens and Steps 2-6 are automated.
- **DataOpsCI.yml** – The YAML file that guides the orchestration and contains important environmental variables.

Step 2 – The automated scripts identifies any new or changed .pbix files and publishes to the Build workspace using a service account with a Power BI Premium Per User (PPU) license.

Step 3 – Any new or changed .pbix files are refreshed (if they are a dataset). *NOTE: The first time a new file is placed in the Build workspace, the refresh will fail.* You have to "prime" the pipeline and set the data source credentials manually. As of August 2020, this is not fully automatable and the Power BI team at [Microsoft have recently written about it](#). My instructions on GitHub call this out as well.

Step 4 – Any tests sitting in the same folder as each .pbix file are ran against the .pbix file in the Build Workspace using XMLA (which is why a service account with PPU is needed). Please note that regardless of if the .pbix file is new, changed, or unchanged, tests are executed. Failed tests are logged to Azure DevOps and the build will fail.

Step 5 – If a .pbix file has all the tests pass, then the .pbix file is moved to the Development workspace.

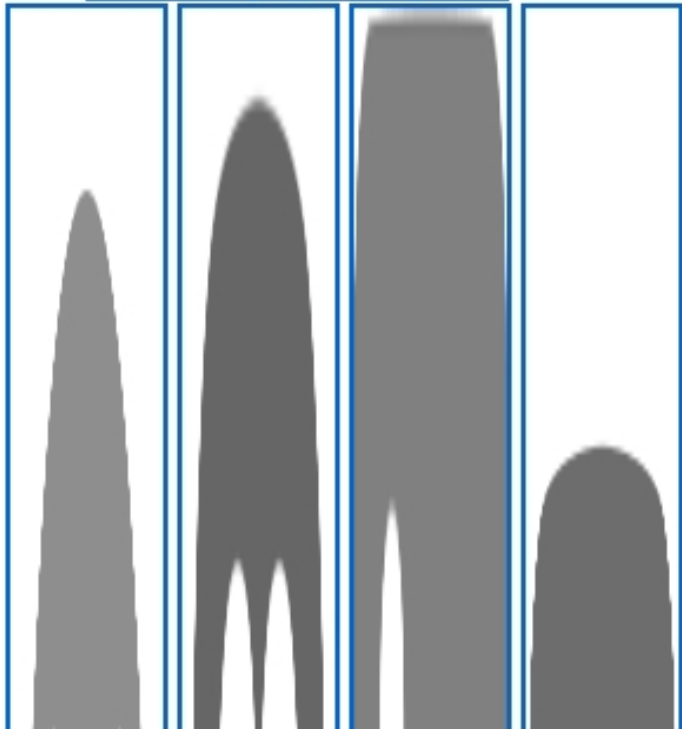
Step 6 – If step 4 for a .pbix file has resulted with a failed test, then the build will fail and the Development workspace will not be updated.

The scripts I have [shared on GitHub](#) utilizes Azure DevOps for version control and CI. If you're a GitHub Actions user, it probably won't be that difficult to convert and if you're interested in writing instructions about implementing this with GitHub Actions, message me on [LinkedIn](#) or [Twitter](#).

You're probably wondering why I left out Test and Production workspaces! well just like I did with Part 3 and Part 4, there is some additional best practices I want to share before completing the whole picture (yep the teaser for you to keep coming back to this blog).

Previous [Part 4: Bringing DataOps to...](#)

Next [Part 6: Bringing DataOps to...](#)



Part 6: Bringing DataOps to Power BI

24 Aug 2021 4 mins read

DataOps Principle #13: Simplicity

We lacked consistency a senior developer tells me this after he reviewed some of the Power BI datasets we were building. We [had tests](#), we [were versioning](#), so progress had been made, but the naming conventions of tables and measures varied slightly. If data analysts/customers were going to use our datasets we didn't want them looking at two different datasets and seeing different names (DateDim, DimDate, Date Dimension, etc.) for the same dimension. And for the senior staff, we didn't want to spend hours applying good design with manual reviews. In short, we wanted predictability, standardization and a simple way to enforce that.

DataOps Principle #13: We believe that continuous attention to technical excellence and good design enhances agility; likewise, simplicity – “the art of maximizing the amount of work not done” is essential.

However, introducing simplicity when it comes to applying consistent design with datasets does have its challenges in Power BI:

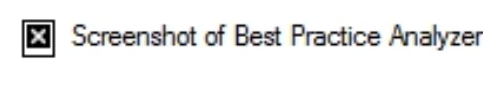
The Challenges

- Power BI is flexible** - We can name our tables, measures, columns anything we want. As a developer, I like flexibility because it allows for creativity and the ability to adjust to customer needs. As a manager though, the thought of the development team building models for the same customer but have them look completely different makes me break out in hives. And it is further distressing that I must ask senior staff and myself to waste time (time not delivering value to customers) enforcing standardization through manual reviews and subsequent rework with updates to datasets.
- There are no built-in linters** - Never heard of a linter? A linter performs static analysis on your code to look for bad coding style/practices, forcing you and your team to keep a consistent code style. There are [linters for many programming languages](#), but Power BI inherently lacks a linter.

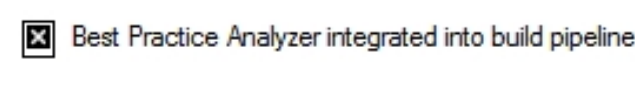
Best Practices

First and foremost, I'm standing on the shoulder of a giant here. [Daniel Otykier](#), the author of Tabular Editor, and [his blog series](#) around Analysis Services allowed these best practices to be viable in Power BI.

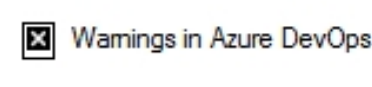
- Leverage the [Best Practice Analyzer](#) in [Tabular Editor](#). Version 2.0 of Tabular Editor is free to use, and the best practice analyzer leverages a JSON file to define rules on DAX expressions, Error Prevention, Formatting, Maintenance, Naming Conventions and more. And in the spirit of simplicity, others have already created JSON templates with predefined best practices such as [Microsoft](#) or [Daniel Otykier](#).
- Just like I suggested that your team use [DAX Studio to test Power BI](#) on their local machine, your team can use Tabular Editor to run your Power BI dataset through the Best Practice Analyzer. That way you start conditioning your team to check for issues that go against best practices prior to committing the Power BI dataset to Git and fix those issues.

 *Example of Best Practice Analyzer and, in this case, the model has many issues*

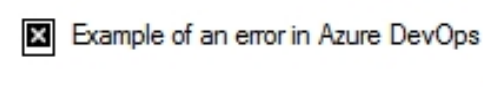
- Incorporate the Best Practice Analyzer in your continuous integration pipeline. Now I made it simple here, because I took the pipeline template from Part 5 and [added a step that runs the Best Practice Analyzer to Github](#). The pipeline now behaves like this:

 *Updated Pipeline with Best Practice Analyzer in Step 4*

If warnings (as indicated with Severity Level 2 in the JSON file) are encountered, the build doesn't fail but you will see the warnings listed in Azure DevOps. An example of a warning is leaving key columns visible in the model.

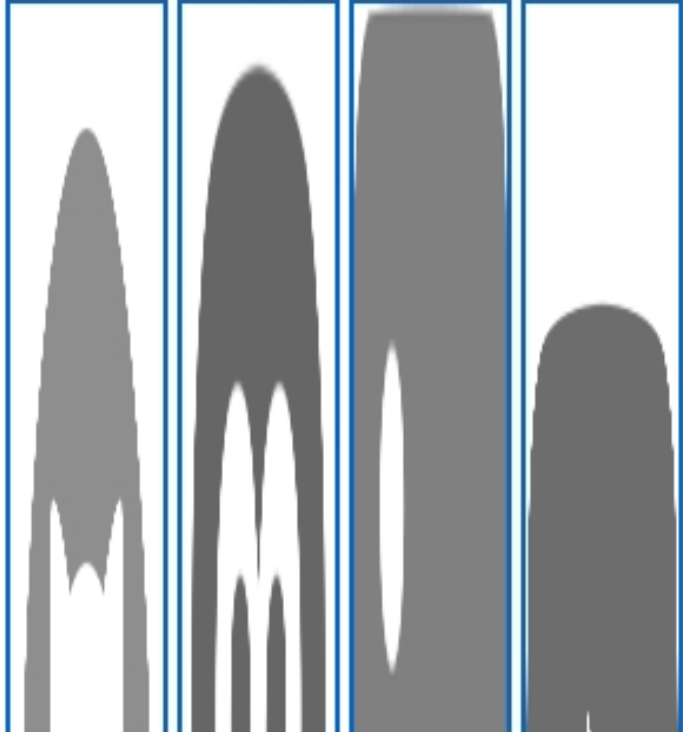
 *Example of warnings from the Best Practice Analyzer in Azure DevOps*

Errors that are encountered will fail the build, and Azure DevOps will list those errors. An example of an error would be to use a variable/measure in the denominator for division instead of using the DIVIDE function which handles the divide by zero issue.

 *Example of an error from the Best Practice Analyzer in Azure DevOps*

Let me know what you think on [LinkedIn](#) or [Twitter](#), and stayed tuned for my next article that will cover best practices in embracing change with Power BI. It won't be so technical this time (fans of Azure DevOps and automation will have to wait a couple more weeks!)

Previous [Part 5: Bringing DataOps to...](#)
Next [Part 7: Bringing DataOps to...](#)



Part 7: Bringing DataOps to Power BI

31 Aug 2021 5 mins read

DataOps Principle #3: Embrace Change

Change is constant and the Power BI team at Microsoft has taken this idea and accelerated it. I've been working in the Power BI field for over 6 years and the pace of new features can be overwhelming. Now, I bet there are some cynics who would say the change/feature they want hasn't come quick enough, but at the macro level the pace of change is tremendous.

DataOps Principle #3: We welcome evolving customer needs, and in fact, we embrace them to generate competitive advantage. We believe that the most efficient, effective, and agile method of communication with customers is face-to-face conversation.

In my experience, keeping up with the Power BI changes is key to meeting customer needs. When features like Paginated Reports and XMLA dropped into General Availability (GA), they became game changers. Had our team been too late in offering these features as a part of solutions, we would have missed the chance to deliver added value to our clients. *But with the litany of changes every week in Power BI, and the work you still need to deliver, how do you keep up?*

Here are some of my thoughts on managing the pace of change:

Best Practices

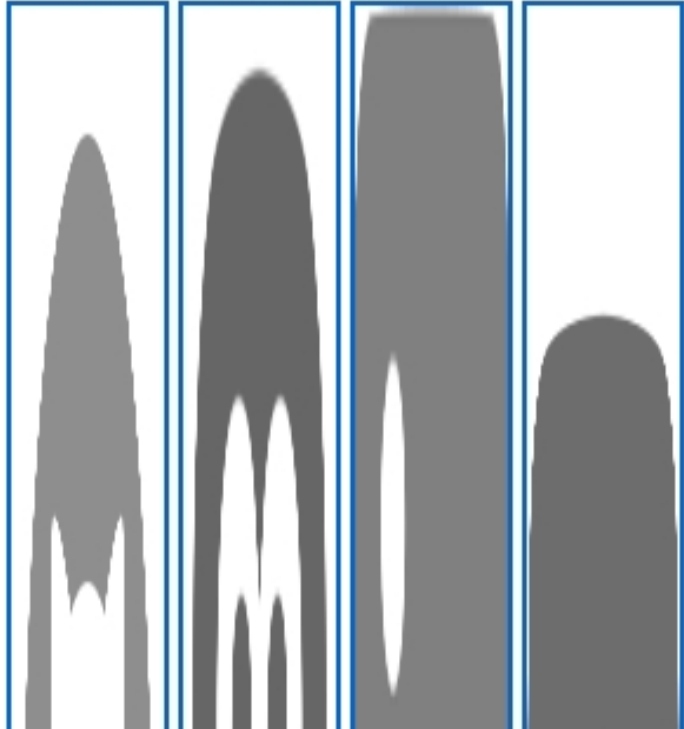
- Follow the release schedule pattern** -- Power BI news is constant, but it also has a pattern. After 2 years of following this schedule, you can keep up with the news and still get work done:
 - Every Thursday afternoon (roughly after 6 p.m. Eastern):*
 - Check the [Power BI blog](#) as the team will have announced most of the week's news by this time.
 - Every Week:*
 - Catch up on monthly Power BI releases and expert opinions on the [BiFocal Podcast](#). [John White](#) and [Jason Himmelstein](#) know what they're talking about and if they are excited about a new feature, pay attention!
 - Check out [Guy in a Cube YouTube videos](#). If you've been working in the Power BI space for more than 3 months, you should be following [Adam Saxton](#) and [Patrick Leblanc](#).
 - Also watch any new videos on [Curbal's YouTube channel](#) I have come to rely on [Ruth Pozuelo Martinez's](#) videos for great analyses on Power BI updates and DAX tutorials.
 - Every Month:* There is a wealth of great blogs in the community, and I've found these very helpful to check for updates:
 - [SQLBI](#) -- I wish I was half as smart as [Marco Russo](#) and [Alberto Ferrari](#). These gentlemen are geniuses, excellent teachers, and post great content.
 - [RADACAD](#) -- [Reza Rad](#) and [Dr. Leila Etaati](#) provide insightful tips on Power BI and I found their modeling, visualizations, and DAX articles great reads. In fact, if I didn't read [Dynamic Row Level Security with Profiles and Users in Power BI : Many-to-Many Relationship](#), I probably would still be having performance issues with Row-Level security.
 - [Excelerator BI](#) -- [Matt Allington](#) seems to have a new article that addresses an issue or new feature I am investigating. His article on [Affordable Power BI Premium for Small Business](#) is my favorite and was updated recently.
 - [Chris Webb's Blog](#) -- I probably reference [Chris'](#) blog articles on M more than any other blog and he provides such in-depth explanations.
 - Every 3 months:*
 - Keep up with [Microsoft's major events](#) because they usually release big news during these events. If you can't watch the virtual event or attend in person (someday hopefully), they usually post a [Book of News](#) that will allow you to catch up.
 - Check out the [Microsoft Roadmap site](#) and view the release plan. This will provide you a forecast of upcoming changes to Power BI (or at least the ones they are willing to share ahead of time).

Side note: I am not implying that you shouldn't listen to any of these superb professionals on a more frequent basis, these are simply my own guidelines to balance between work, life, and the pace of change.
- Empower Others** -- You don't need to know everything, don't try to be the hero. In fact, if you're on a team, empower others to follow parts of the schedule I outlined in the previous best practice. Have them report back to the team on findings and discuss ways to incorporate features in your team's schedule.
- Know when to use Preview versus General Availability** -- If Microsoft says the feature is in preview, use that as the time for prototyping and experimentation. Only when it is in GA should you consider the feature production-ready.
- US Sovereign Cloud is Different** -- The final best practice is for those working on the US Sovereign Cloud (GCC, DoD). Often the unstoppable force of Power BI updates will meet the slow-moving object of government. For some features, they are available in the government clouds as soon as they are GA in the Microsoft commercial cloud. For others, you may be waiting some time for [government approval](#) before it shows. To keep up to date on what is available and/or how to use a new feature in government be sure to check Microsoft's [US Government Business Applications Demo Repository](#) and [Feature Availability](#) sites. I'd also recommend subscribing to [Steve Winward's YouTube Channel](#) for specific tutorials on using Power BI and Power Platform capabilities in the US government clouds.

Let me know what you think about these best practices on [LinkedIn](#) or [Twitter](#). For those who wanted more Azure DevOps and automation content, stayed tuned for my next article. In that article I'll discuss the automation of testing metrics for Power BI.

Previous [Part 6: Bringing DataOps to...](#)

Next [Part 8: Bringing DataOps to...](#)



Part 8: Bringing DataOps to Power BI

14 Sep 2021 5 mins read

DataOps Principle #8: Reflect

Iâ€™m on a ship with no compass, thatâ€™s how I felt about our testing. Our team had done a decent job implementing the [testing of our Power BI datasets](#), but how could our team reflect on its operations without knowing if we are testing at acceptable levels? Or worse, if someone was adjusting the tests and accidentally reduced the amount of testing being applied.

As a project manager I was tasked with building dashboards to facilitate a clientâ€™s ability to reflect on their performance and track their progress towards a goal. Was I a hypocrite if I couldnâ€™t do the same with my teamâ€™s development of the dashboards? It sure felt like it.

DataOps Principle #8: Analytic teams should fine-tune their operational performance by self-reflecting, at regular intervals, on feedback provided by their customers, themselves, and operational statistics.

The Challenge

In software testing, code coverage and its related metrics allows teams to reflect on the quality of the products being developed. But low-code tools, like Power BI, do not inherently let you evaluate the tests of M code or DAX code as you would expect. For example, take the DAX measure:

IF([Sum of Sales]>64,â€Greenâ€,â€Redâ€)

If I could hypothetically convert this DAX measure and the tests weâ€™ve written to JavaScript, I could use tools (like [Istanbul](#)) to inspect the tests of the code and get answers to the following questions:

- Function Coverage â€“ Did I test that DAX measure?
- Statement Coverage â€“ Did I test every line of that DAX measure?
- Branch Coverage â€“ Did I test the second argument in that IF statement of that DAX measure?

Best Practices, sort of.

So far this blog article has asked several questions and provided no answers. Well, my answers are a work in progress, and I feel like what I have built is something between [MacGyver](#) and [Rude Golberg](#). But Iâ€™ll let you decide. This has worked with a few of my Power BI projects, and our team can now reflect on our testing and we can set goals.

Low-Code Coverage Tool

I use the term Low-Code Coverage, because my solution, without true access to the underlying code, cannot fully calculate function, statement, or branch coverage and meet the definition (in my opinion) of code coverage. But I think it can get close.

The tool consists of 5 major components:

- 1) Tests â€“ Pretty obvious but very important. If you have read [Part 5](#) of this series, you should have the templates and best practices to build tests for tables, rows, and calculated columns for the dataset.
- 2) [Tabular Object Model](#) â€“ The tool needs to be able to update a Power BI dataset and the Tabular Object Model offers the easiest methods to accomplish this in PowerShell. Thanks to [Marco Russo's article](#) and [Kevin Marquetteâ€™s article](#) for their inspiration and guidance on this topic.
- 3) [Dynamic Management Views](#) (DMVs) â€“ Available in premium datasets via XMLA, the tool will use DMVs to get the following metadata information about the dataset:
 - a. Calculation Dependencies (\$System.DISCOVER_CALC_DEPENDENCY) -- This table allows us to figure out what measures, tables, and calculated columns are used in the tests created.
 - b. Tables (\$System.TMSHEMA_TABLES) -- This table identifies the tables in the Power BI dataset.
 - c. Measures (\$System.TMSHEMA_MEASURES) -- This table identifies the DAX measures in the Power BI dataset.
 - d. Columns (\$System.TMSHEMA_COLUMNS) -- This table identifies the calculated columns in the Power BI dataset.

DMVs provide us critical information on what measures, tables, and columns are used in our tests and what measures, tables, and columns exist in the dataset. With this information we can calculate Low-Code Coverage metrics.

4) Azure DevOps â€“ I expect the Low-Code Coverage data to be collected for each build of the Power BI dataset, and Azure DevOps will facilitate this need.

5) SharePoint â€“ We need to log Low-Code Coverage data and SharePoint will allow us to track progress with Low-Code Coverage metrics over time. I chose SharePoint because I thought it would be convenient for others to use. If youâ€™re using Power BI and Azure DevOps, there is a good chance you have SharePoint Online available.

Low-Code Coverage Tool in Motion

Alright enough about the components, you probably want to know how it all works in Azure DevOps. Here is the workflow:

☒ Low-Code Coverage Azure DevOps Pipeline example *Low-Code Coverage Workflow*

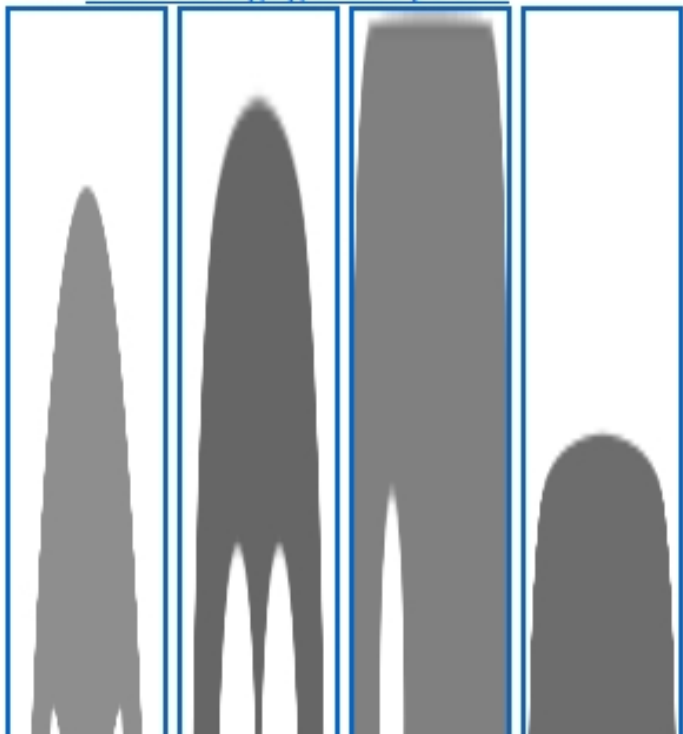
Letâ€™s look at each step:

- Step 1** â€“ Youâ€™ve updated the tests for the Power BI dataset and now pushed the changes to Git on Azure DevOps.
- Step 2** â€“ Any .pbix file within the PBI project folder is cloned and published to the Build workspace with a unique name. I wanted to make sure the original Power BI dataset in the build was immutable and would not be impacted by this process.
- Step 3** â€“ Any tests sitting in the same folder as each .pbix file are added as a calculated table using the Tabular Object Model. This does require it to be installed when the build agent in Azure DevOps runs, and this will add to the time it takes the pipeline to complete. You can look at custom build agents to save on build times (sounds like a future blog post)
- Step 4** â€“ The tests are now in the cloned dataset, so we can query the DMVs to get the information we need for Low-Code Coverage.
- Step 5** â€“ With the query results retrieved, we log them to a SharePoint list that can store the query results and build ID.
- Step 6** â€“ The cloned dataset has served its purpose and is deleted to clean up the Build workspace.

I feel like that was a lot to unpack in one article, so I have created a [proof-of-concept in GitHub](#) that will run Steps 1-6 on an Azure DevOps pipeline. Please let me know what you think on [LinkedIn](#) or [Twitter](#).

In my next article, Iâ€™ll show how you can take the data logged to SharePoint and build Low-Code Coverage metrics with Power BI.

Previous [Part 7: Bringing DataOps to...](#)
Next [Part 9: Bringing DataOps to...](#)



Part 9: Bringing DataOps to Power BI

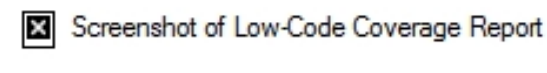
21 Sep 2021 4 mins read

DataOps Principle #8: Reflect (cont.)

“What gets measured gets done” I’ve seen that message numerous times over my career and, along with DataOps, has influenced the way I look at and improve upon team performance. In [the last blog article](#) I explained Low-Code Coverage and detailed how we can log the underlying data it needs using Azure DevOps, PowerShell, and SharePoint Online. Now that we have the data, we need to extract it from SharePoint and transform it into something us mere mortals can measure as Low-Code Coverage sounds like a job for Power BI!

DataOps Principle #8: Analytic teams should fine-tune their operational performance by self-reflecting, at regular intervals, on feedback provided by their customers, themselves, and operational statistics.

So, let’s start at the end so you can see the final product first:

 *Figure 1 “Screenshot of the Low-Code Coverage Report”*

This type of report has allowed my teams to reflect on their testing and have a Low-Code Coverage goal to achieve. In the example above, that goal is 80% and you’ll notice that in this case the KPI reads 70% meaning we haven’t achieved the goal. The KPI is a DAX measure that is built by the two numbers that are displayed to the right of the KPI: Components with Tests (the numerator) and Components Requiring Tests (the denominator). Each is calculated through the following high-level steps:

Components Requiring Steps - Leveraging the Measure, Table, and Column data extracts from the Dynamic Management Views, Power Query parses the XML for each extract and combines them into one table. Each row in the table identifies the type of component, Table, Measure, or Calculated Column. It also contains a concatenated key that combines the build ID, dataset name, component name, and component type separated by a pipe character. With that schema, we can use a distinct count to calculate the denominator by build, or by dataset name.

Components with Tests - Now that we know the tables, measures, and calculated columns that need testing we can leverage the Calculation Dependencies extract to identify what components were used in the tests. Within Power Query, I parse the XML of the Calculation Dependencies and extract just the DAX measures representing the tests. To make it easier, I prefix the name of each test in the DevOps pipeline to distinguish the tests from other DAX measures. We then build the same concatenated key as we did for the denominator and then left join the denominator table with the numerator table. This allows us to identify the rows that have tests, those that don’t have tests, and build some custom columns to make it easier to calculate the numerator. *As an aside to Microsoft, I would like to be able to merge tables using more than one column please, the concatenated key approach is getting a little old.*

Alright so if you were not able to follow that, I loaded [the Power BI template into GitHub](#), so you can review the step-by-step Power Query I used to build the dataset. If you do look in Power Query, I do want to point out the following:

1) SharePoint API Query “The out-of-the-box connector to SharePoint is slow, so you will see 4 functions that helps query SharePoint Online through the REST API. We can then just get the columns we need for the dataset and reduce the time to refresh the dataset.

 *Figure 2 “Custom Power Query functions to get SharePoint data (faster)”*

2) Test Bridge Table “There is a many-to-many relationship between the components and the tests, so I followed the [Kimball approach and built a bridge table](#).

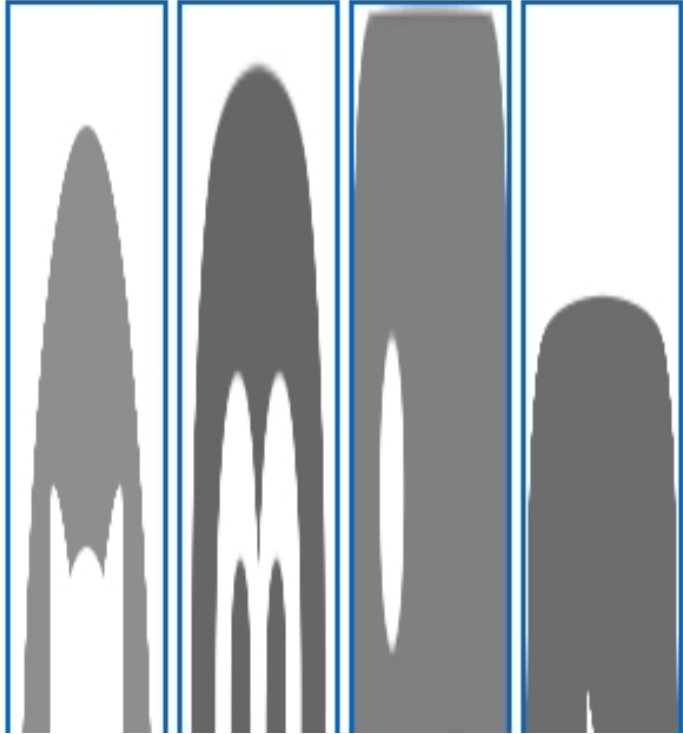
3) Build Information “This hidden table is used to help me get the Power BI Dataset name for the concatenated key when building the denominator. Unfortunately, the Measure, Table, and Calculated Column schemas don’t identify the dataset name. However, the Calculation Dependencies schema does have it, so we can use the SharePoint List ID to fetch that information.

4) Dates “The SharePoint REST API sends dates back in UTC, so the Created Date field in the dataset is in UTC. If you need to adjust for time zones, I suggest modifying that column. The Power User has a [good blog article](#) on the topic of time zones.

If you’re not interested in using the template but want to see the Low-Code coverage report in action, I published it to the web [at this link](#). Please let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 8: Bringing DataOps to...](#)

Next [Part 10: Bringing DataOps t...](#)



Part 10: Bringing DataOps to Power BI

05 Oct 2021 6 mins read

DataOps Principle #14: Analytics is Manufacturing

“Patience is a virtue” I am not a patient person. Red lights at traffic stops vex me when there is no cross traffic. If I see long queues at restaurants, I’m more likely to avoid the place all together because I don’t want to wait. I have a lot to learn about patience when I cannot control the situation. On the other hand, at work where I do have some control, I can focus my impatience on process improvements!

DataOps Principle #14: Analytic pipelines are analogous to lean manufacturing lines. We believe a fundamental concept of DataOps is a focus on process-thinking aimed at achieving continuous efficiencies in the manufacture of analytic insight.

The Power BI development lifecycle and the Azure Pipelines that support continuous integration and deployment is also analogous to manufacturing lines. If you’ve tried implementing [Part 5](#), [Part 6](#), or [Part 8](#) of my blog you may have noticed the checks I put in place to install PowerShell modules, Tabular Editor, and Analysis Management Objects (AMO) when needed. That’s because as of October 3rd, 2021, the images available as a [Microsoft-hosted agent](#) do not have all the dependencies pre-installed to run the pipeline. This appears to lengthen each run because certain installations must occur before further steps in the pipeline can execute. But how much more time is taken dealing with these dependencies and installations?

The Questions

Thankfully, Azure DevOps offers the ability to create [self-hosted agents](#) to install the prerequisites and adjust processing power to see if efficiencies can be gained. Many reading this may say, well of course a properly configured self-hosted agent would run the pipelines quicker. I would tend to agree with you, but that does not necessarily follow the principles of DataOps. We need to show empirically that efficiencies are gained and demonstrate a return on investment in self-hosted agents. As a result, we will gather data to answer the following questions:

1. How much faster can a self-hosted agent run versus a Microsoft-hosted agent?
 - a. If we can create a self-hosted agent with 7GB of RAM and 2 virtual CPUs ([the standard for a Windows image on a Microsoft Hosted Agent](#)) how much faster does it run?
 - b. If I increase RAM or virtual CPUs by 50% is there a gain in performance?
2. Is the cost for running a self-hosted agent cheaper than having staff not wait for pipelines to run?
 - a. If so, how often must the pipeline run to realize savings?

By answering these questions, a team can confidently move forward or pull back on pursuing self-hosted agents.

The Experiment

The architecture layout for this experiment is illustrated in the Figure below. The Resource Group, Containers, Container Registry, and Docker are highlighted in yellow to show the newest components to the layout since [Part 5](#) of this series.

 Architectural Layout *Figure 1 “Components of the experiment for self-hosted agents”*

I chose to use Azure and containers for hosting and running self-hosted agents respectively. My familiarity with Azure influenced me to choose this cloud platform, but if you’re interested in recreating it in Amazon, Google Cloud, or another cloud, please reach out to me as I would love to see the results when the cloud platform variable changes.

In the [GitHub repo](#), I provide step-by-step (hopefully) instructions to run these tests, but I would like to provide the following details/comments on a few components:

Resource Group “The resource group hosts the resources we need to create the self-hosted agents and is located within the same region where the Azure DevOps instance is located. Since network latency is a variable, I tried to mitigate variability (since I can’t physically touch the network cables) by keeping the self-hosted agents in the same location.

To find where your Azure DevOps instance is located please follow [these instructions](#).

Docker “Using the [Microsoft instructions](#) as a basis, Docker is used to instruct images to install the prerequisites for running an Azure Pipeline and install the PowerShell modules, Tabular Editor, and Analysis Management Objects.

Container Registry “Once the Docker image is created, the image definition should be hosted in a registry in order manage the Docker images. This experiment follows DataOps principles so if self-hosted agents are worthwhile, we should treat Docker images as code and make them easier to manage/reuse.

Container Instances “Also known as the self-hosted agent, this is where most of the pipeline work is done. For the purpose of the experiment, there are two container instances, at 7GB RAM/2vCPU agent to try match the Microsoft-hosted agent, and a 10.5GB RAM/3vCPU agent with 50% more memory/processing resources.

Experiment in Action

The experiment runs three different agents, the Microsoft-hosted agent, the self-hosted agent with similar resources (7GB RAM/2vCPU) and a self-hosted agent with 50% more resources (10.5GB RAM/3vCPU). Each agent runs the Continuous Integration job that includes [Part 5](#) and [Part 6](#) of this series as well as the Low-Code Coverage job from [Part 8](#). To gather data, the YAML file that runs the pipeline is scheduled to run every 20 minutes. After a 24-hour period we should have enough data to analyze and answer questions.

Learning What Not To Do

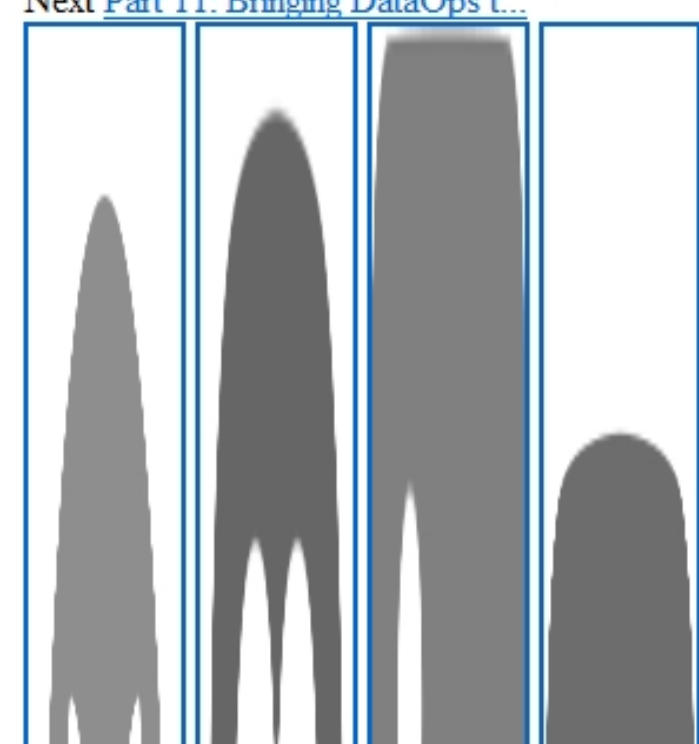
To be honest, the pursuit of these answers was not easy. I spent more time than I care to admit learning several ways not to do things and the one way that worked consistently (relatively). With that said, I’d like to share a couple of roadblocks I hit so that you may learn from my mistakes, or at least not feel discouraged if you encounter them yourself.

- 1) Setting up Docker on a Windows machine was frustrating. It was only after setting up a Windows Server following [these directions](#) and going the Hyper-V route (instead of [WSL 2](#)) did I successfully create the self-hosted agent.
- 2) I had to create a mix of Azure CLI, Docker, and PowerShell commands to make this effort reproducible. This was because an API may have a feature that another API didn’t have, or one API was easier in implementing a task than the others. For example, I ended up calling the Azure DevOps REST API via Invoke-WebRequest because the [Azure DevOps extension](#) for Azure CLI [did not have a command to create agent pools](#). In addition, scripting the creation of containers was far easier in PowerShell than Azure CLI. I felt a little like Dr. Frankenstein grafting together different APIs to build a monster codebase.

Alright, so if you missed it above, the instructions to run this experiment can be found at [this link on GitHub](#). In my next article, I’ll share how to take the data stored on Azure DevOps and come to some conclusions leveraging Power BI (I can’t help myself). And if that isn’t exciting enough, the next article will provide examples of how to query Azure DevOps’s API using Power Query, so you may find that topic interesting. I won’t have any tips on improving your patience because I’m still working on that. In the meantime, please let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 9: Bringing DataOps to...](#)

Next [Part 11: Bringing DataOps t...](#)



Part 11: Bringing DataOps to Power BI

21 Oct 2021 6 mins read

DataOps Principle #14: Analytics is Manufacturing (cont.)

Time is money Continuing our work from [Part 10](#) we have the data to help us understand (1) if self-hosted agents run faster than the Microsoft-hosted agent and (2) if we get a return on investment using a self-hosted agent by not paying staff to wait for a pipeline to run.

DataOps Principle #14: Analytic pipelines are analogous to lean manufacturing lines. We believe a fundamental concept of DataOps is a focus on process-thinking aimed at achieving continuous efficiencies in the manufacture of analytic insight.

To do this in Power BI involved a little bit of research. First, I wanted to avoid inputting any data manually, so that it could be templated for other projects. That meant I had to figure out how to work with the Azure DevOps and Azure Pricing APIs.

Azure DevOps API

First, either the documentation had some issues or my reading comprehension was failing me (a possibility). Regardless, I had three issues:

1. In order to pull data into Power BI I used the Personal Access Token (PAT) to access the API. The PAT had to be Base64 encoded, but I was completely unaware you had to prefix the PAT with a semicolon *before* Base64 encoding. Thank goodness Stack Overflow had the answer (what would I do without it?) because the Microsoft documentation sure didn't help me.
2. I had no clue how to Base64 encode in Power Query. Thanks to [Dale](#) on the Power BI Community support page, I found the right incantation. So, as a result, this was the Power Query statement created:

```
Binary.ToText(Text.ToBinary(":" & #"Personal Access Token"), 0)
```

1. As of October 2021, there is no way in the Azure DevOps API to discover what [Azure region your Azure DevOps instance is running on](#). I wanted to be slick and automatically pull the region information so it could be used to filter the Azure Pricing API by region. That was a big NO ugh.

After that, I used Web.Contents to query the [Pipeline](#) and [Builds](#) API endpoints and that was straightforward. If you're not familiar with Web.Contents, I suggest reading [Chris Webb's article](#) as it is the best documentation out there.

Azure Pricing API

Have you've ever tried to figure out cloud pricing and felt confused? Do you like pain? If so, then try to interpret the [Azure Pricing API](#) and the JSON that is returned. Trying to figure out the right combination of 'serviceFamily' and 'serviceName' for Container Registry and Container Instance pricing took countless rounds of attempts in [Postman](#). On top of that, I wanted to get all the Azure regions from the Azure Pricing API since issue #3 with the Azure DevOps API existed. I couldn't find that endpoint either (double facepalm). I settled with just getting a distinct list of regions from the Container Registry pricing call. Not perfect, but it works.

Whew! that was probably a lot of complaining but for me that was truly cathartic. Plus, I offer you this [template on GitHub](#) so you don't have to be bitter like me.

Putting it together

With the dataset in place, it was time to get the answers into some visuals. From experience, I found an executive summary highlighting the return of investment (ROI) resonated far more with a boss than a chart. That's why the template Power BI report looks like this:

☒ Executive Summary example

The Executive Summary tab has three drop-downs:

1. Azure Region - Select the Azure region where your [Azure DevOps instance is located](#).
2. Build Agent - Select between the two versions of the container instances created from [Part 10](#).
3. Hourly Rate - For this template you have the choice of selecting between \$0 and \$200 an hour which will determine the minimum number of runs before you realize a return investment with running a self-hosted agent.

By selecting those three drop-downs, Power BI measures located in the Measures Table will take:

- The 'Average Time Saved Each Run For Build Agent in Minutes' and multiply it by the 'Minute Wage for Staff' to get the 'Potential Average Cost Savings Per Run'.
- Takes the 'Potential Average Cost Savings Per Run', runs through the hypothetical scenarios of 0 and 200 runs, and identifies the lowest amount of runs required to be greater than the monthly cost to run the self-hosted agent. This was accomplished using the RANKX function in 'Rank Cost Avoidance By Number of Runs'.

The Results Are In!

As for my recent project, the executive summary answered the questions from Part 10 as follows:

Question 1A: How much faster can a self-hosted agent run versus a Microsoft-hosted agent? If we can create a self-hosted agent with 7GB of RAM and 2 virtual CPUs ([the standard for a Windows image on a Microsoft Hosted Agent](#)) how much faster does it run?

Answer 1A: On average, we saved 2.37 minutes per run.

Question 1B: How much faster can a self-hosted agent run versus a Microsoft-hosted agent? If I increase RAM or virtual CPUs by 50% is there a gain in performance?

Answer 1B: On average, we saved 2.45 minutes per run.

Question 2: Is the cost for running a self-hosted agent cheaper than having staff not wait for pipelines to run? If so, how often must the pipeline run to realize savings?

Answer 2: For confidentiality purposes, I won't divulge the hourly rate I used, but the answer was yes and the number of monthly runs required to achieve ROI was realistic.

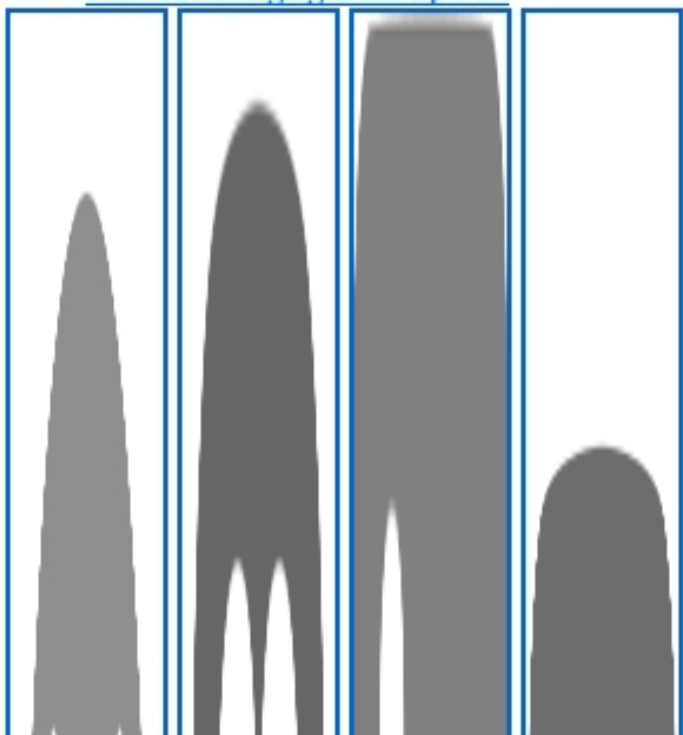
Conclusion

For those that went through Step 10, the Power BI template and the setup instructions for Part 11 have been posted [on GitHub](#). And if you want to try out the Power BI report using data I generated, try it out [at this link](#).

Despite the trials and tribulations of working with the APIs, in the end we were able to empirically show whether or not a self-hosted agent would improve efficiency, saving time and money! As always, if you have any thoughts on this article or corrections/solutions to the issues I had with the APIs, please me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 10: Bringing DataOps t...](#)

Next [Part 12: Bringing DataOps t...](#)



Part 12: Bringing DataOps to Power BI

08 Nov 2021 9 mins read

DataOps Principle #7: Reduce heroism

“You saved the day!” one of our clients told my teammate. We had attended a meeting with our clients and their bosses reviewing one of the latest Power BI reports we had deployed. Important decisions were being made with the data, and one of my teammates had the foresight to make a copy of the underlying Power BI dataset and store it before it is refreshed. Their thinking was that questions may arise about the data at the time those decisions were being made; therefore, the client needed a reliable way recall the data. Naturally two weeks later, questions from other leadership started circulating concerning the decisions that were made and the data used to make those decisions. When our clients consulted us, our prophetic teammate spoke up and provided a link to the copy of the data on a OneDrive folder. Our teammate deserved a cape and capital P (for Power BI of course) emblazoned across their chest a hero.

I was very thankful for our teammate swooping in to save the day, but just like a popular animated series about heroes, I couldn’t help but wonder *what if?* What if that teammate was not present at the meeting? What if our team wasn’t around to make that back up?

DataOps Principle #7: As the pace and breadth of need for analytic insights ever increases, we believe analytic teams should strive to reduce heroism and create sustainable and scalable data analytic teams and processes.

After that event, I talked with our team to discuss how we could empower our client to run their own backups. First, we looked at the parameters we were working in:

- 1) Office 365 E3/Premium Per User Environment – We had to operate within the bounds of the current licensing setup, and additional resources like Premium or Azure were not possible. Also at the time, [Backup and restore datasets with Power BI Premium](#) had just entered preview and if you read [Part 7 of my series](#) that option was not yet production ready.
- 2) Ease of Use – We need to give our clients the ability to kick off a backup and easily retrieve it so it could be integrated in their environment.
- 3) Import Only – Our dataset was developed using the import mode, so we knew all the data was contained with the dataset as a snapshot in time.

Second, we looked at our options, and decided on a route: Power Automate and SharePoint. We could create a flow that our client could run and using [Power BI’s Export to File API endpoint](#) we could let flow run the backup and save it to a SharePoint site managed by our client.

Since I came from the SharePoint world before moving to the Power BI world, I volunteered as tribute to create the *Backup Flow*.

The Setup

Having worked with Microsoft Flow before it became Power Automate, I knew the HTTP connector that we could use was not available under the current licensing model. Back in 2019, Microsoft had introduced a wonderful challenge in my life by moving [that connector to premium status](#). I spent weeks explaining a convoluted licensing framework to clients, and in the end I just rearchitected several solutions due to a lack of funding anyway. So, now that I am older and wiser (thanks to those wonderful challenges) I knew that a better route was a [Custom Connector](#) and not the HTTP connector. If you’re interested in how to setup a Custom Connector, there have been many blog articles written on the topic. [Konstantinos Ioannou’s article](#) and [Chris Webb’s article](#) are two that I have bookmarked and refer to for guidance.

After half-a-day, I had a flow that looked like this:

 Example Backup Flow

And it worked! We tested it with our production datasets and didn’t run into any issues and after a few days the clients were trained on how to use the Backup Flow.

Blog article done, right? Nope, I just had to adhere to DataOps principles and sure enough I ran into some issues. Issues I was glad I ran into before our clients did.

Making it Reproducible

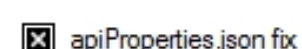
With this principle in mind, I wanted to make sure the Custom Connector and Flow could be stored in version control and make it easier to deploy to other environments (dev, test, prod). So, I ventured into the world of using [Microsoft Power Platform CLI](#) to deploy the Custom Connector. And that’s where the trouble started. I needed to install Python, just another prerequisite to the already growing list of prerequisites needed to script deployments. I found myself using the self-declared APP stack (Azure CLI, PowerShell, Python) to script out the following:

- 1) Create an Azure App Service Principal with the appropriate delegated permissions to Power BI.
- 2) Update the Custom Connector’s apiProperties.json file with client id of the new Azure App Service Principal.
- 3) Deploy the Custom Connector using the CLI. This includes providing a client secret from the Azure App Service Principal.

Step 3 was where I really hit a wall. This one-line of code was my roadblock for several days:

```
paconn create -s "settings.json" -r $CredentialResult.password
```

After determining that it may not be my fault, I went to the GitHub page for the CLI code and [found this issue](#). Thanks to [Nirmal Kumar](#) I was informed of a work around that involved a minor change to the apiProperties.json file:

 apiProperties.json fix

Did you see the change? A lowercase ‘a’ instead of an uppercase ‘A’! Arrrgh!

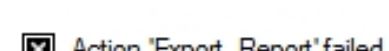
I was very thankful for the fact that the open-source community allowed for information sharing, but equally frustrated that this wasn’t tested. It seems like such an important part of the tool.

With that roadblock out of the way, I finished the script. I loaded [the script on GitHub](#) and [the Flow on GitHub](#) as well. I don’t have any readme instructions, and we’ll get to that reason shortly.

Monitor quality and performance

From my experience with SQL backup jobs, I knew that if not properly monitored a backup job could easily fail and go unnoticed, or take longer and longer to complete over time causing data pipelines to be impacted. The current Backup Flow worked, but how would it fair if I increased the size of the data model?

I then set forth with a small dataset, increasing it by about 10 MB and running the Backup Flow. At around 20MB the Flow started to fail with this message:

 Action ‘Export_Report’ failed

So, my inclination was valid, and I quickly warned our client of the possible failure and work around (export via Power BI Service). Now I needed to find out why the timeout occurred. It turns out a [Custom Connector has a two-minute connection timeout](#). Alright, so this whole Backup Flow idea works up to a point and when you reach around 20MB for a dataset, you’re out of luck with this option. Thus, no readme instructions because I wouldn’t necessarily recommend this approach. This was a learning experience for me, and I hope you learn from my cautionary tale and become wiser without getting older.

PreferClientRouting

Note: As of November 16, 2021 the issue in this section does not appear on the East2 tenant (hurray!), but I will keep it in the blog for sentimental purposes.

What did we end up doing to replace the Backup Flow? I thought Power BI PowerShell would allow us to come to the rescue. But when I tested [Export-PowerBIRestMethod](#) with large files, I would get a 500 internal server error. I then tried the Invoke-PowerBIRestMethod route using the [Export Report in Group](#) and got the same 500 error. Alright, as a sanity check I tried to download the pbix files I was testing using the [manual method](#). They worked just fine (head scratcher).

It was now time for me to read some documentation. Mercifully, at the top of the documentation for Export Report in Group there was this message:

Note: As a workaround for fixing timeout issues, you can set preferClientRouting to true. Large files are downloaded to a temporary blob. Their URL is returned in the response and stored in the locally downloaded PBIX file.

After digging a little deeper, if I used the preferClientRouting parameter there should be a 307 redirect with a location header identifying the temporary URL for the pbix file.

So I appended ‘preferClientRouting’ to the API call via Invoke-PowerBIRestMethod and got a 403 Forbidden error. I then turned to Google and GitHub to see if anyone has faced this issue; no luck. At this point I reached out to folks on Twitter and I posted an issue on [Microsoft’s Power BI PowerShell GitHub](#). Since I’m stubborn and impatient, I didn’t wait long to break out Postman to see how the Power BI API was behaving. Then I opened developer tools on Chrome, and looked to see what network requests occurred when we tried downloading a pbix through the [manual method](#).

What I have discovered is the redirect URL provided by the API in the 307 response is inaccessible. However, the URL that works via the manual method is at the same host (ex. https://wabi-us-east2-b-primary-redirect.analysis.windows.net) but at a different endpoint ‘export/v201606/reports/{Report ID}/pbix’. Okay, now we’re getting somewhere.

After a couple hours I update the PowerShell using the new endpoint (which works with the bearer token) and successfully test downloads up to 100 MB. Shazam!

Note: This seems to be only an issue on the commercial Microsoft tenants I’ve tested and, yes, I have opened a support issue at <https://support.powerbi.com>.

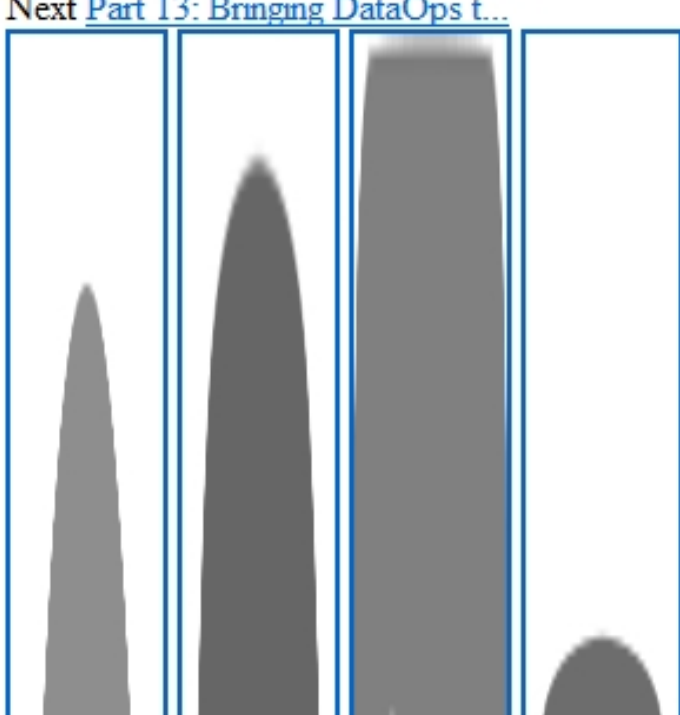
Conclusion

After some instruction and SOP updates, our clients could run this PowerShell themselves and we were more confident this process would work. So, I’m content to say that *after they got the hero they wanted, they got a process they deserved* (oh you knew that was coming).

If you’re interested in downloading the PowerShell script it is [available at this link](#). As always, if you have any thoughts on this article or corrections/solutions, please let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 11: Bringing DataOps t...](#)

Next [Part 13: Bringing DataOps t...](#)



Part 13: Bringing DataOps to Power BI

23 Nov 2021 7 mins read

DataOps Principle #17: Reuse

“Don’t lose it, reuse it!” is a catchphrase from a popular show that my little one watched obsessively for over a year, and that catchphrase has been seared into my brain. Unsurprisingly, that catchphrase would repeat in my head after reviewing several Power BI datasets. Our team had created several [custom functions in Power Query](#) and they were being used across datasets. While I was pleased to see custom functions were being embraced, ensuring that these functions were being properly reused across datasets and managed was whole other ordeal. Within the first two months of our team embracing custom functions, we had spent several hours copying updated M code for custom functions into various datasets. This was an unacceptable use of their time, and we needed a methodology to rescue us from the copy-and-paste firestorm that lurked just around the next update.

DataOps Principle #17: We believe a foundational aspect of analytic insight manufacturing efficiency is to avoid the repetition of previous work by the individual or team.

The Challenges

As I have growled before, sometimes bringing patterns involving high-code (in this case [code reuse](#)) to a low-code tool can be challenging. Applying the reuse principal was not an exception, and we quickly identified some challenges:

- 1) Version Control “ While the custom functions were a part of the dataset and were being versioned using the techniques in [Part 3](#), the custom functions themselves represented something that could be used across datasets. How could we make sure staff were using the appropriate custom functions and avoid having to manually inspect each dataset for differences in the code? And how do we avoid copy-and-paste updates?
- 2) Package Management “ A lot of our custom functions could be logically grouped together like global M functions (Table, Text, Number, etc.). Was it possible to do that with a group of custom functions? Was that the right approach?
- 3) Package Dependencies “ If we could solve the version control issue, how could we make sure datasets upgrade to different versions of the custom functions on our own schedule? For example, if I had version 1.0 of Custom Function A running on Dataset B, how would I make sure when version 2.0 of Custom Function A is released that Dataset B does not automatically switch to 2.0 on the next data refresh without testing?
- 4) Documentation “ While the name of the custom function could be self-explanatory, how could we make it easier for our team to understand the parameters and possible considerations when using a function?

The Research

With all the questions in front of us, I scoured articles on the sites of the Power BI community I knew and tried to sniff out patterns they used to *reuse* custom functions. Throughout this section I’ll try to reference who provided these patterns, but I will admit these references are not exhaustive and those that I reference may or may not be the original provider of a pattern.

Version Control

This was an easy pattern to identify; Git was the solution. I first came across Curbal’s video for [Reusing Functions](#) and quickly found that other community members like [Maxim Zelensky](#), [Imke Feldman](#), and [Igor Cotruta](#) stored their Power Query functions in Git.

Package Management

Throughout my search, I found three patterns of packaging custom functions:

Function import “ from library import function_x, function_y

With this pattern you only bring in the custom functions you need from Git. More specifically, each Power BI file has a custom function that fetches the code from Git, a “load” function. For example, [Kiara Grouwstra’s](#) library has a LoadFunctionFromGithub.pq function that will pull another custom function from GitHub and add it to the model.

LoadFunctionFromGithub

Figure 1 - LoadFunctionFromGithub Power Query Snippet

The downside to this approach is the function is named with a default value (Invoked Function) and you still need to rename the function.

Module import “ from library import *

With this pattern you bring in all the functions using a similar load function, and then you choose which ones to add by clicking the function link and selecting “Add as New Query” (see Figure 2). I came across this pattern from Kim Burgess’s [m-tools library](#).

Add as New Query

Figure 2 “ Choose a function from a whole library

The nice thing about this approach is the name of the function is automatically added, so no manual renaming is required!

.mez Maneuver

I found this approach from [Eugene Meidinger’s](#) article and [Igor Cotruta’s](#) library and it involves using the PowerQuery SDK to build a Custom Connector that packages the custom functions. This pattern involves compiling the custom functions into zip files with a .mez extension (who names these extensions?). To leverage locally, you need to have the .mez files in the Custom Connectors folder. Unfortunately, this presents a fun message:

Uncertified Connectors pop-up

Figure 3 “ Yikes, that’s not a fun message.

There is a way to turn it off, but if you have security reviews on your Power BI projects this may require some explaining.

Data Extensions Setting

Figure 4 - Not Recommended? This should be fun to explain to our InfoSec team.

To leverage in the Power BI service, you also need to have a [gateway with the custom connector installed](#). For some projects, getting a gateway installed or convincing admins to install a custom connector may be a non-starter.

Package Dependencies

If you have worked with yarn or pipenv, then this concept is not new to you, but I’ll provide a brief explanation. Let’s say you have a custom function in the master branch of your repository with two parameters Foo(x as text). You want to improve Foo by adding another parameter y and with your new logic this parameter is not optional. If you were to update the master branch with the new Foo you would break any Power BI report using this function on their next refresh and see a message like this:

Error Message

So how do you prevent this? I didn’t see many examples out there, but if we use Git and the Function/Module import pattern, then the Load function could be updated like this:

LoadFunctionFromGithub with version

The branch parameter allows you to define which version of the code to import. Consequently, you could have a parameter in your model that defines what branch to use, update the parameter, fix anything that breaks, test, and redeploy your model confident that the new version of the custom function works.

Side note: If you see the #shared keyword in the code above and don’t know what it means, don’t fret, Imke Feldman and Lars Schreiber are on the case and have [an excellent series explaining #shared and the environment concept](#).

Documentation

Very early on in my career I was told that documentation and code comments was like cleaning the guest bathroom. It’s a pain to upkeep, but others will surely appreciate its cleanliness when they visit. So how could you properly comment your custom functions? Well, I didn’t know this was even possible with custom functions until I saw this Imke Feldman function:

A nicely documented function

This is a great pattern to emulate and you can see how Imke accomplishes this by looking at [an example here](#).

My Thoughts

While there are several options with both pros and cons depending on your environment, I settled with this pattern for my typical projects moving forward (until a better pattern or patterns emerges):

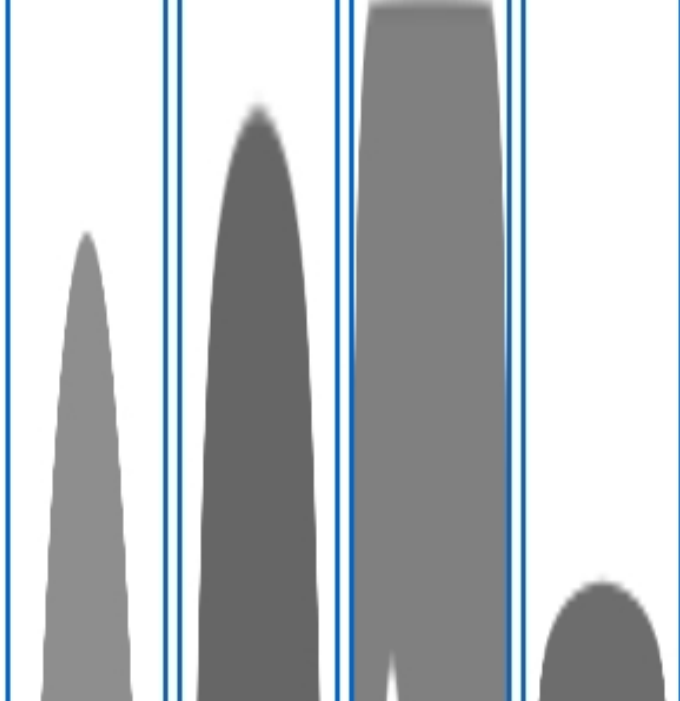
- **Version Control:** Git
- **Package Management:** Module import “ from library import *
- **Package Dependencies:** My own flavor of the Load function
- **Documentation:** Imke’s pattern.

Conclusion

I tried to exemplify the patterns I found in my research in a Power BI file that is [available for download at this link](#). As always, if you have any thoughts on this article, have a pattern I didn’t see, or corrections/solutions, just yelp for help on [LinkedIn](#) or [Twitter](#).

Previous [Part 12: Bringing DataOps t...](#)

Next [Part 14: Bringing DataOps t...](#)



Part 14: Bringing DataOps to Power BI

11 Jan 2022 5 mins read

DataOps Principle #9: Analytics is code

Now how did that get there? A client asked me about a new column that had appeared in a report. I quickly checked the file and sure enough, the new column was there. I was surprised to see the column, and while surprises are fun for birthday parties, surprises are not my idea of fun at work. I looked at the data lineage and traced it back to a SharePoint list. I was not an administrator on that SharePoint list, but a new column had been added to that list recently and it managed to creep into our model. I talked with the client and removed the column from the model (since it wasn't required for the reporting).

The issue at hand was resolved, but I knew our team needed a way to avoid this kind of issue in the future. We had been monitoring for refresh issues with Power BI datasets as explained in [Part 2](#), but this type of bug was different because our Power Query code didn't fail on refresh, but rather introduced a column we did not plan for from a source we did not maintain. We needed a way to identify schema changes before it appeared in production.

DataOps Principle #2: Analytic teams use a variety of individual tools to access, integrate, model, and visualize data. Fundamentally, each of these tools generates code and configuration which describes the actions taken upon data to deliver insight.

In this situation every time a dataset was refreshed, we introduced new data into the model. Based on this principle, we needed to shift our mindset - every time a dataset was refreshed, we introduced new code into the model . And that code needed to be tested. But there are some challenges:

The Challenges

1) **Testing Schema Differences** Under "Scheduled Refresh" in the Power BI Service's dataset, there is no checkbox that if checked would "Fail Refresh If Schema Changes". If such a checkbox existed a team could be warned of a failed refresh, and the dataset would not introduce untested code.

☒ If only there was a "Fail Refresh If Schema Changes" checkbox

Figure 1 "If only there was a "Fail Refresh If Schema Changes" checkbox

2) **Setting Expectations on the Content** - Expanding on Challenge #1, there is no built-in way to monitor for **data drift** (dun dun dunnnn!). Data Science readers may know this term, but for those unfamiliar with data drift, it is the unexpected and undocumented changes to data structure and semantics. For example, let's say Column X in Table A was a string column with 20% null values when you deployed it to production for the first time. Over time, the underlying source system caused that string column to report more null values to the point that it contained 50% null values; would you want to know that? I sure would want to know that, and I would definitely want to test for that trend *before* my customers see it. Or, better yet, I could be transparent and present data drift to my customers (on Power BI of course) so they could understand data quality issues.

3) **There are no built-in linters** You may be asking "Wait a minute John, you mentioned this before in [Part 6](#)?" Yes, and Tabular Editor does a great job with evaluating DAX expressions, metadata, model layout, and naming conventions, but does not evaluate the Power Query code (I hope I'm wrong, please someone tell me there is a way to do this in Tabular Editor). More specifically, if we could evaluate our code for say [Table.SelectedColumns](#), we could make sure that certain tables explicitly define the columns that would be used from the source.

A Probable Solution (Work Around)

Alright, so to address Challenge #1, we need to design for a continuous deployment pipeline that (1) conducts a refresh of a Power BI dataset in a staging workspace, (2) tests for schema differences, and (3) "deploys" to the production workspace if no differences are found. Figure 2 illustrates the pipeline, and I will note that this assumes you're using import mode for the dataset (sorry live connection or direct query folks, but feel free to contact me if you're trying to do something similar).

☒ Continuous Deployment of a Power BI dataset using import mode

Figure 2 "Continuous Deployment of a Power BI dataset using import mode

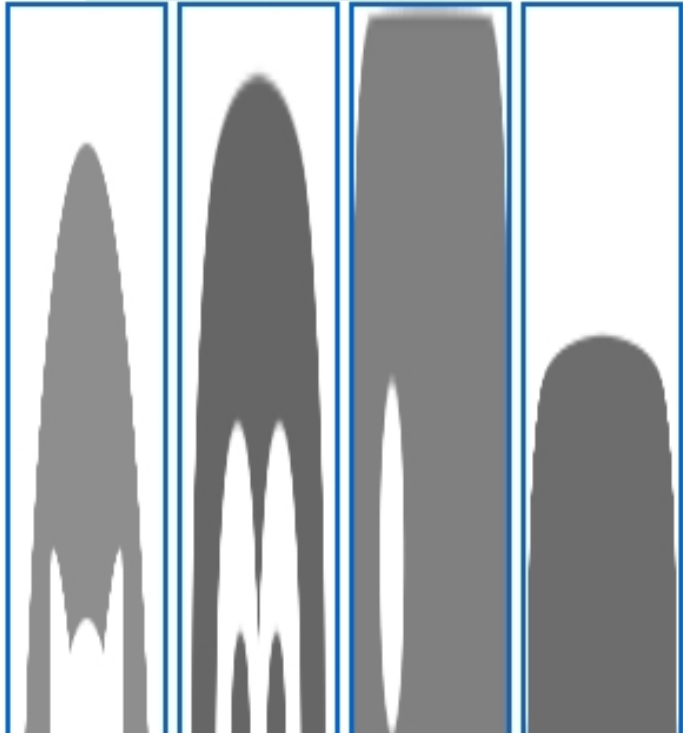
Thanks to Azure DevOps we can build such a pipeline based on the previous work we did (see [Part 5](#)) to do the following:

- 1) Refresh Dataset in Staging Under "Using Power BI's PowerShell" we refresh the Power BI dataset.
- 2) Test Refreshed Dataset Using Tabular Editor's awesome capabilities, we can run a script in the Azure DevOps build agent to extract the schemas from the dataset's staging version and product version. With the two schemas extracted, we can then compare them for differences. If the schema differs then we issue an error in Azure DevOps (which fails the job). Otherwise, we move to step 3.
- 3) Refresh Dataset in Production With confidence that the schema has not changed, we issue a refresh to the production dataset using PowerShell.

I have an example pipeline with install steps shared on [GitHub](#). Please let me know what you think on [LinkedIn](#) or [Twitter](#). Stay tuned for my upcoming articles that address challenges #2 and #3 as well (yep, another teaser).

Previous [Part 13: Bringing DataOps t...](#)

Next [Part 15: Bringing DataOps t...](#)



Part 15: Bringing DataOps to Power BI

15 Feb 2022 5 mins read

DataOps Principle #9: Analytics is code (cont.)

And now we address Challenge #2. In [Part 14](#) I identified the challenge of setting expectations on a refreshed dataset in Power BI. More specifically how do we look for the issues of **data drift**, the unexpected and undocumented changes to data structure and semantics, in our production datasets?

Testing, Testing, Testing!

If you have read through the series, you may have noticed I am a big proponent of testing. If my co-workers were quizzed on what topics I like to talk about the most, there is a high probability that one of the topics will be "testing" (the second would probably be DataOps). So my proposed solution includes (drum roll) Testing! (to the surprise of no one). Let me elaborate on *how* we should test though.

Using the scaffolding from [Part 5](#) we can update the folder structure so that we can distinguish the tests we have built during development (CI) from the tests to identify cases of data drift (CD).

☒ Folder Structure for testing data drift

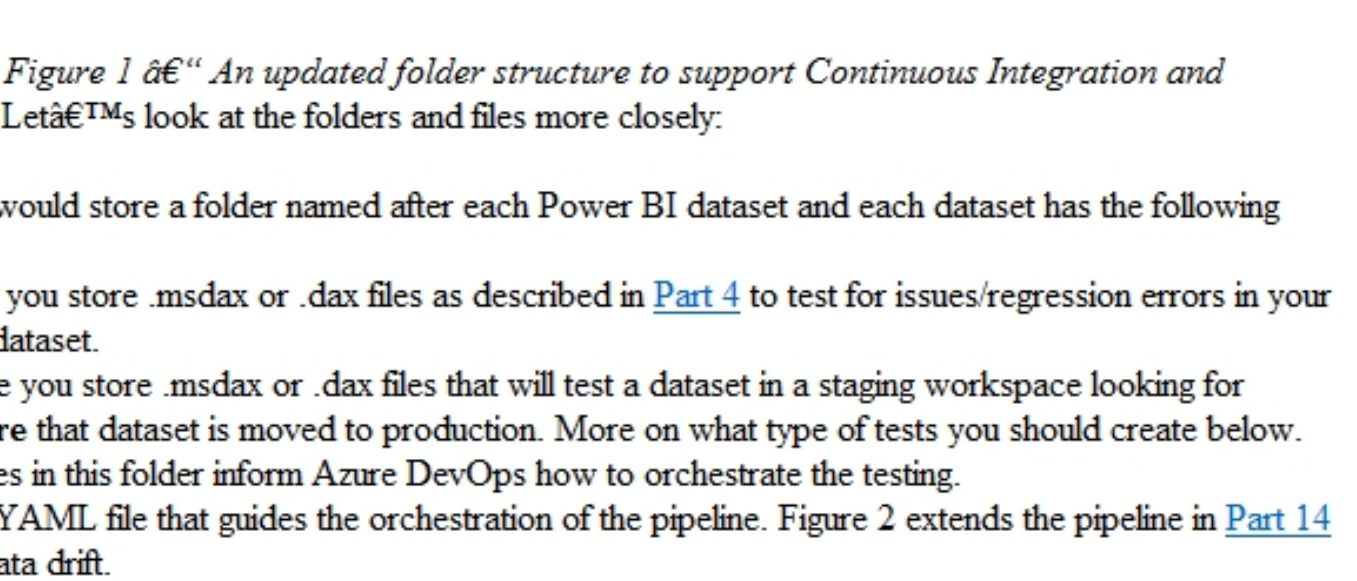


Figure 1 "An updated folder structure to support Continuous Integration and Continuous Deployment testing" Let's look at the folders and files more closely:

- PBI** "This is where you would store a folder named after each Power BI dataset and each dataset has the following subfolders:
 - CI** "This is where you store .msdax or .dax files as described in [Part 4](#) to test for issues/regression errors in your model using a static dataset.
 - CD** "This is where you store .msdax or .dax files that will test a dataset in a staging workspace looking for potential issues **before** that dataset is moved to production. More on what type of tests you should create below.
- PipelineScripts** "The files in this folder inform Azure DevOps how to orchestrate the testing.
- DataOpsCD.yml** "The YAML file that guides the orchestration of the pipeline. Figure 2 extends the pipeline in [Part 14](#) to now include testing for data drift.

☒ Pipeline for Continous Deployment



Figure 2 "The updated Continuous Deployment Pipeline"

What to test

With the folder structure and pipeline in place, how do we test for data drift using DAX? First, I recommend using a schema like I described for automated tests in [Part 4](#). Second, I focus on testing columns within each table in a dataset using the following categories of tests:

- Primary Keys/Distinct Values** "Nothing can throw off a model's behavior more than when a column value you expected to be unique, isn't unique. In order to test for that I've found the two DAX functions, COUNTROWS and DISTINCTCOUNT, to be helpful. Using these two functions we can calculate how many rows exist for table that has that column and then calculate the distinct values for that column. Here is an example:

```
// Row Count
VAR _Row_Count =
    COUNTROWS ( 'Test Table' ) + 0
// Id has distinct values
VAR _Id_Distinct_Count =
    DISTINCTCOUNT ( 'Test Table'[Id] ) + 0
```

Then it's a matter of testing if the values are equal. If not, we have a problem.

- Data Has Acceptable Values** "This can take a couple of forms: 1) *Verifying a column has no null columns* "The example below calculates the number of null values in the Title column. A test case would fail if this variable is greater than 0.

```
// Calculate if blank values exist in the title column
VAR _Title_Is_Not_Blank =
    CALCULATE (
        COUNTROWS ( VALUES ( 'Test Table'[Title] ) ) + 0,
        ISBLANK ( TRIM ( 'Test Table'[Title] ) )
    )
```

- *Verifying a column has acceptable values*** "If you have a known list of values, you can calculate how many rows have this acceptable set of values. The example below checks for acceptable values using the IN command and also includes allowing blank values to occur (which can be included or excluded depending on the need). If the Row Count for the table does not match the result of this DAX expression, then the test case would fail.

```
// Test for acceptable values
VAR _Acceptable_Values_Example =
    CALCULATE (
        COUNTROWS ( 'Test Table' ) + 0,
        FILTER (
            VALUES ( 'Test Table'[New Column] ),
            'Test Table'[New Column]
                IN { "A", "B" }
                || LEN ( TRIM ( 'Test Table'[New Column] ) ) = 0
                || ISBLANK ( 'Test Table'[New Column] )
        )
    )
```

Wondering about using a Regular Expression to test the contents of columns? Well, as of February 2022, that is not built into DAX and should be discussed at a later time (I smell a future blog article!) 3) *Verifying a column is in an acceptable range* "This form is helpful for a numeric or date column. For example, if you expect dates to not be in the future, then you can build a test to make sure the source systems don't supply future dates without a test highlighting the problem.

```
// Dates are less than today
VAR _Dates_In_Acceptable_Range =
    CALCULATE (
        COUNTROWS ( 'Test Table' ) + 0,
        FILTER ( VALUES ( 'Test Table' ), 'Test Table'[Date Received] < TODAY ( ) )
    )
```

- Acceptable Null Percentage** "Sometimes a column may contain null values, but you may need to monitor if the percentage of null values increases to an unacceptable threshold. You don't want a critical column in a dimension to increasingly contain null values without your knowledge! In the example below, the DAX formula counts the number of blanks and divides by the row count using the DIVIDE function (better than using the '/' operator because row count could be 0 and lead to a divide by zero issue).

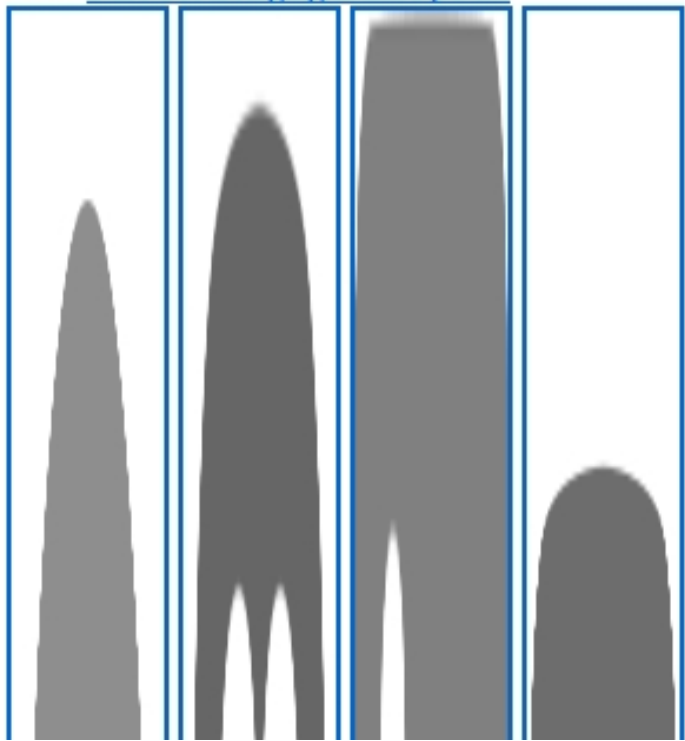
```
// Null Percentage
VAR _Null_Per_Example =
    DIVIDE ( COUNTBLANK ( 'Test Table'[Null Example] ) + 0, _Row_Count, 0 )
```

You don't have to start from scratch because I've built examples of each aforementioned category of tests for a simple Power BI dataset along with an updated pipeline that mimics the flow in Figure 2. Instructions for downloading and installing this sample is available [on GitHub](#).

What are your thoughts on this approach? Could my DAX test cases be improved? Please let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 14: Bringing DataOps t...](#)

Next [Part 16: Bringing DataOps t...](#)



Part 16: Bringing DataOps to Power BI

16 Mar 2022 4 mins read

DataOps Principle #9: Analytics is code (cont., cont.)

I love Regular Expressions but I've been told I may lean on them too much. In fact, in the past I would ask syntax-based regular expressions questions during interviews in order to evaluate potential candidates. Now that I'm older and hopefully wiser, I've learned that a question such as "what is the difference between the '\$' or '^' character in an expression?" is unproductive and does not determine whether or not a candidate would be suited for data engineering. In fact, I've worked with several exceptional colleagues over the years who could not recall specific regular expression syntax in an interview. Yet they knew when regular expressions (I'm calling Regex from now on) were the right tool to use and how to build an expression quickly using well know accelerators.

So, what does Regex have to do with Power BI? Well, from my colleagues and other experts in the data industry, I've learned that Regex is an essential tool to identify data drift within data pipelines. However, building Regex in the Power BI space is challenging:

The Challenges

1) **No built-in support** " If you type "Regular Expressions in Power BI" in your search engine of choice, you'll find some excellent articles by [Shivam Shukla](#) and [Philip Seamark](#). Yet these articles don't cover Regex support with the Power Query language (M) or DAX. These articles explain how to use the R language, but this requires you to install R on your desktop and requires you to learn yet another language. Don't get me wrong, R is a great language and a nice option for Power BI, but the two core languages of Power BI don't support Regex.

2) **Not production-ready** " So if you're left using R (or Python for that matter) for running Regex in Power BI, then you're left with a big problem if you want to run this code in the Power BI Service. [You need a personal gateway installed](#) to schedule a refresh; furthermore, as of March 2022 you cannot use an enterprise data gateway (much to the consternation of several folks on the [idea forum](#)) or the Power BI service to schedule a refresh. In my experience, if you can't use an enterprise data gateway or the service, the R/Python option is not viable for long-term success.

A Probable Solution (Work Around)

I've been experimenting with this approach for a few months now and this approach allows you to run Regex expressions against Power BI datasets leveraging the pipeline from [Part 15](#). This approach involves the following components:

- PowerShell** " Yes this is another language, however, it is not one you need to learn because it's written into the pipeline and uses the built-in Regex libraries to run the expressions.
- Regex Health table** " Each row contains the data that defines how to inspect the contents of a column using a Regex. I chose this format because the alternative was storing a separate file in the project folder that would have a tabular schema. Instead of introducing a new file format or schema, I felt having a table within the dataset define the Regex checks would be easier to manage and take up less space in the model. The Regex Health table consists of 3 columns:
 - Table - The table name as precisely defined in the model.
 - Column - The column name as precisely defined in the model.
 - Regex - The expression using the [.NET format](#).

 A hidden table in the model to track Regex

Figure 1 " A hidden table in the model to track Regex
- Power BI API** " I've been waiting for the right time to take advantage of the newer feature of [executing DAX queries over the Power BI Rest API](#). With this feature we can pull the contents of the Regex Health table but also pull the contents of the columns.

With these components you can then check the contents of a dataset and detect potential data drift with Regex using a flow as illustrated in Figure 2.

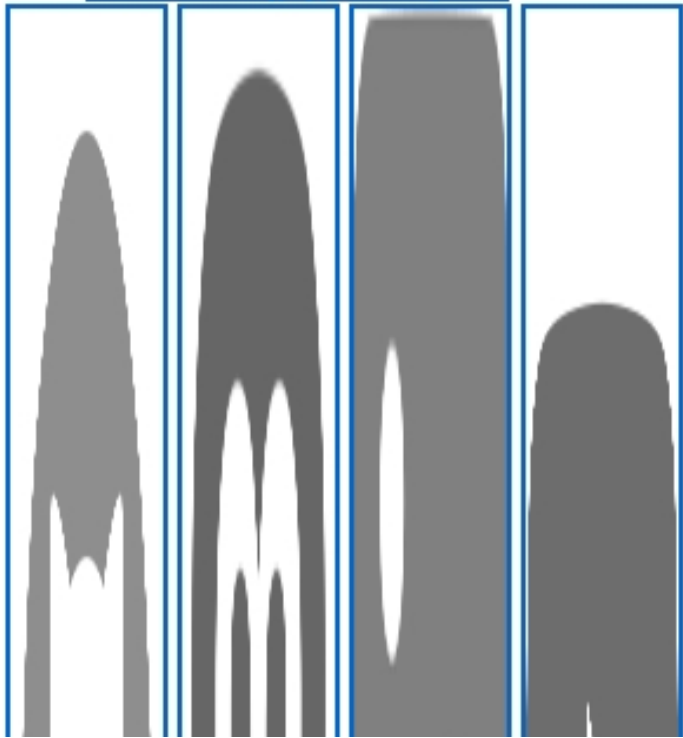
-  Flow for detecting data drift with Regex
- Figure 2 " Flow for detecting data drift with Regex

This flow is encapsulated into a single PowerShell function [Confirm-RegexPassesWithPPU](#) and added to a preexisting pipeline as defined in Figure 3. Now the pipeline will run the Regex tests after all the DAX tests ([see Part 15](#)) have completed.

-  A new continuous deployment pipeline with Regex checks
- Figure 3 " A new continuous deployment pipeline with Regex checks

I'd really like to hear your feedback on this possible solution, so I've loaded this pipeline, installation instructions, and a sample Power BI dataset [on Github](#). Try it out and let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 15: Bringing DataOps t...](#)
Next [Part 17: Bringing DataOps t...](#)



Part 17: Bringing DataOps to Power BI

18 Apr 2022 7 mins read

DataOps Principle #1: Continually satisfy your customer

It's complicated that's how I would sum up my relationship with SharePoint. On the one hand, SharePoint has helped me deliver modern solutions to clients and helped me advance my career. On the other hand, SharePoint has led me to numerous late nights and elevated blood pressure levels due to features like the list view threshold, lookup threshold, and CAML. Honestly, if it were not for folks like [Marc Anderson](#), [Julie Turner](#), [Andrew Connell](#), [Laura Rogers](#), and [Steve Sanderson](#) and their content on how to make the most out of SharePoint, I would have broken up with SharePoint a long time ago.

I have had to learn to work with SharePoint, so when I was [monitoring a Power BI model](#) that heavily relied on SharePoint list data and saw it was taking a dramatically longer time to refresh, I took a deep breath, and took the time to understand how to work with SharePoint to improve the refresh time and keep the customer satisfied.

DataOps Principle #1: Our highest priority is to satisfy the customer through the early and continuous delivery of valuable analytic insights from a couple of minutes to weeks.

With this principle in mind, I needed a way to make sure the client could get the data from SharePoint into Power BI as quickly as possible. I did not want automated refresh jobs to take 30 minutes (which is what was happening!). My initial theory was the connector and/or the way the data was being pulled from SharePoint Online was inefficient. While working with the SharePoint Online List Connector it was apparent that all the columns, even the ones not needed, were being pulled into the model. Working on the theory, I needed to look at the options to query the SharePoint data with two main objectives:

- 1) Only query the data you need. Superfluous columns from SharePoint increase the size of the response over the network. The more data, the more time needed to retrieve that data.
- 2) Reduce the calls to SharePoint. The less calls to SharePoint to get the data, the less time to refresh Power BI.

With those objectives in mind, I looked at the options:

Option #1 " SharePoint Online List Connector 2.0 (Beta)

Released in [October 2020](#) as beta (non-production) and over a year later it is still in beta (as of April 2022), this connector provides you an option to pull all the column values or the columns in the default view of SharePoint.

☒ Option 1, the Beta option

Figure 1 " Option 1, the Beta option

This connector has two issues: 1) It is in beta and not production ready according to Microsoft and 2) if you use the default view option, what happens if that default view changes? I did not manage the SharePoint sites that provided the data and could not guarantee the owner of a SharePoint list would not change the default view. Despite that, this option would let me isolate the columns needed for the Power BI model.

Option #2 " OData Feed

The [SharePoint REST API](#) is based on the OData protocol and Power BI offers an [OData Connector](#). With this option I could write a custom set of query string parameters to get the specific columns using the \$select and \$expand query string parameters (objective #1, check). If I used the "Stop=5000" in an OData query, I could retrieve the maximum number of items per call allowed by SharePoint ([list view threshold](#)) and minimize the number of calls to SharePoint (objective #2, check). The only problem is the OData feed [did not support relative URLs](#). Not a massive inconvenience, but I like parameterizing connection information because it tends to make the promotion of models between development, test, and production simpler.

Option #3 " Bespoke

This option is using the SharePoint REST API, but instead of using the OData Connector, I could simply use the [Web.Contents](#) function in Power Query to issue the queries. It is custom, but I found Web.Contents to be an extremely useful function in Power Query and it offered relative URLs. Thus, it was worth a try.

The Test

To see if these any of these options reduced the time to refresh Power BI, I created two SharePoint lists, one with 8 columns and one with 16 columns with a mixture of single-text, numeric, and choice columns. I then loaded each list with 19,999 items (limit for upload with the Excel feature in SharePoint). This closely represented the SharePoint lists I work with, so it was sufficient for testing.

With the list data in place, I began to build the Power BI Reports for the original SharePoint Online Data Connector, Option #1, Option #2, and Option #3. Each version of the report would only retrieve data and not perform any other actions. This way I was hopefully isolating any refresh time to just the time it takes for Power Query to get the SharePoint data. I ended up with 12 permutations of reports. One set of 4 queried the 8-column list, one set of 4 queried the 16-column list, and one set of 4 that queried both the 8-column list and 16-column list.

With the 12 reports complete, I published the reports, verified the credentials in the data settings, and then had to decide how to run the tests. I had some concern that if I ran the refreshes in parallel, I may encounter throttling by the Power BI service or [API throttling by SharePoint](#). Consequently, I chose to run the Power BI Refreshes serially, at least 8 times (rounds), and randomly selected the order in which the reports refreshed each round. I also chose to run 8 rounds of testing under the [license mode](#) for Power BI Pro and Premium Per User to see if refresh times would be different. With the help of PowerShell, I was able to automate these tests.

Finally, I updated the local Power BI monitoring report I shared in [Part 2](#) to track the refresh times for all these reports.

The Results

After over 216 refreshes, the results were in:

☒ Pro Workspace Results

Figure 2 " Pro Workspace Results

☒ Premium Per User Workspace Results

Figure 3 " Premium Per User Workspace Results

I was able to draw two conclusions:

- 1) For out-of-the-box connectors, Option #1 " SharePoint Online List Connector 2.0 (Beta) was far faster than the default SharePoint Online List Connector (1.0) and OData Connector.
- 2) The Bespoke implementation (Option #3) was astonishingly quick.

The function in-depth

I was little surprised that the Bespoke function was the fastest, so lets take a closer look at what it does.

☒ Bespoke Function

Figure 4 " Bespoke Function

The custom function takes three parameters:

- 1) Source " seeded with null and will store SharePoint query results.
- 2) Url " the query string parameters using the OData protocol.
- 3) Count " seeded with a zero, I used this to debug the function because it uses recursion.

The custom function also contains two nested functions:

- _GetJsonFromSharePoint " This function would query the SharePointURL (parameter) with the url parameter that contained the query string parameters
- _GetNextLink " This function would extract the pagination link in order to retrieve the next set of SharePoint data.

The end of the function executes by recursively calling SharePoint for the initial, and any subsequent queries defined by the __next value in the SharePoint response. As the data (in JSON format) is retrieved, it is combined using the List.Combine function. The Result variable then contains a row of JSON for each list item in SharePoint, and that can be parsed by Power Query to produce a table like the other options.

Conclusion

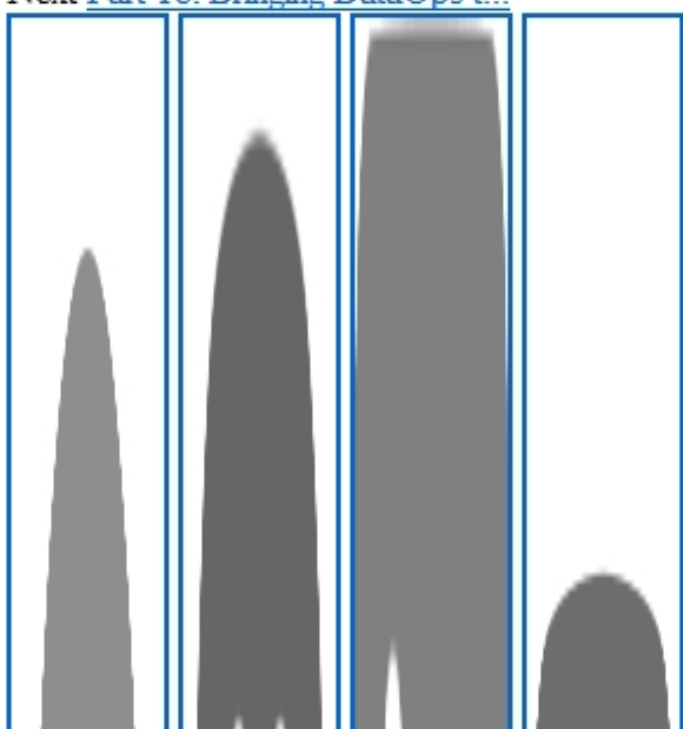
I am interested to see if others could recreate and refine this test scenario. I have loaded the scripts, Power BI reports, and instructions to set up the test [on GitHub](#).

- Am I missing an option?
- Did I make a mistake with how I set up a Power BI Report test?
- Should I reconsider my relationship with SharePoint?

I would really like to hear your feedback, so let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 16: Bringing DataOps t...](#)

Next [Part 18: Bringing DataOps t...](#)



Part 18: Bringing DataOps to Power BI

28 Apr 2022 5 mins read

Distractions, DataOps, and Documentation

If you have read [Part 1](#) of this series, you probably know that I like to listen to podcasts while I go running. To help pass the time, I was happy to listen to a recent DataTalks.club podcast where there was a [recent episode](#) in which [Alexey Grigorev](#) and the audience asked questions of [Chris Bergh](#), the Father of DataOps. This was a wonderful distraction on my run as I listened to Chris explain that teams/organizations should be focused on optimizing from the perspective of the whole system; the data, the model, the visualization, and the governance. DataOps comprises principles of Lean Manufacturing, so those familiar with Lean would call this Optimize the Whole. When we balkanize this optimization, and this is very easy to do when you have large organizations (one in charge of data engineering, one in charge of the business line, one in charge of enterprise architecture, etc.), we risk losing the ability to deliver value because the separate parts consider done/success differently. For the data engineers “done” might be getting the ETL job to complete on time, but are we delivering value when the transformed data is of bad quality?

Near the middle of my run, Chris and Alexey further discussed focus areas when applying the principles of DataOps:

1. Lowering errors in production “Track and prevent errors from the data source all the way to where it creates value, including all the different tools and code that move and transform the data.
2. Reduce cycle time to deployment “Deliver features/updates quicker for the customer.
3. Productivity of the team “Empower your team to get their work done *right*, quicker.
4. Measure “You can’t see progress without tracking items 1, 2, and 3.

This discussion really struck me because earlier in the day I was reading a [new article on advance data model management](#) in Power BI’s implementation planning documentation and I homed in on this visual from the article:

☒ Option 1, the Beta option

Figure 1 “From Microsoft Documentation - Usage Scenario: Advanced Data Model Management

With my DataOps glasses on, I was not able to see where we could show clients/customers how all these relationships are delivering value. I agree that external tools, right now, are required to do advanced data model management, but project teams are often challenged with:

1. Proving that these external tools and their potential license costs deliver value and are worthy investments either in money or the time involved to go through a bureaucratic process of getting them approved and installed.
2. Proving that the code acting on the data, and the data itself is of high quality.

So if I could append this article, I would recommend that when looking at advanced data model management you keep in mind the following based on the items (red circles) from that visual:

Item	John’s DataOps Thoughts
2	• Test the source data for freshness, or if the schema has changed.
Your team should find ways to answer “yes” to these questions:	
4 & 6	• Are you versioning the models? • Are you testing the models? • Are you automating those tests? • Are you logging results of the tests?
	• Are you using a Best Practice Analyzer to make sure your model is sound? • Can you hire a new developer/analyst and be comfortable that any change they make doesn’t introduce regression errors in production?
	• Monitor the gateway and log failures in up-time and failed refreshes.
	• Monitor for issues with data in production, in order to avoid hearing it from customers first. • Log when issues occur so that you can work to address them with your team, data source providers, clients, etc.
15	• Don’t use these logs as a weapon to put down others, but as way to help others improve upon what they manage or have built.
All	• Through containers and scripting you should attempt to automate the creation of your environment to support new projects, experimentation, and staff onboarding; thus, reducing setup costs.

Now it is time for the shameless plug. If you are looking for way to implement the thoughts I presented or the DataOps principles in your Power BI project I have some ideas from [this series](#) and code to share. I know that I do not have all the answers, but I do hope you read these ideas, discuss them with the Power BI Community, and that this eventually leads to 3 outcomes:

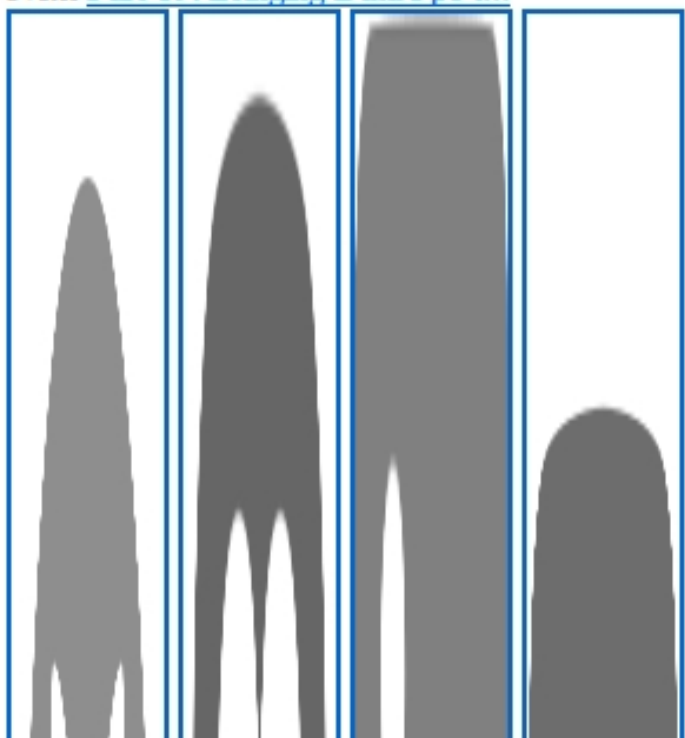
1. Whether you call it DevOps, MLOps or XOps you see how applying the principles in the [DataOps Manifesto](#) will make delivering Power BI solutions better.
2. Because you have better solutions, you spend less time babysitting data models or fretting about issues with Power BI projects and can spend more time with family, take a vacation, or go for a longer run (Reduce heroism).
3. Enough attention is given to the product gaps in bringing these principles to Power BI that they become new features within Power BI and we ultimately make many of my blog articles obsolete.

For #3 I am excited to read recent news that [Power BI reports/datasets will begin to be incorporated into the Power Platform’s Application Lifecycle Management](#) in July 2022. This could be a great first step, and I’ll be sure to share my opinions on this blog.

And if you’re looking for other ideas around this topic, I suggest following [Daniel Otykier](#) and [GreyskullPBI](#) on Twitter if you don’t already.

What do you think? Did I miss the point of the Power BI article? Am I weird for listening to podcasts while I run? Let me know your thoughts on [LinkedIn](#) or [Twitter](#).

Previous [Part 17: Bringing DataOps t...](#)
Next [Part 19: Bringing DataOps t...](#)



[Home](#) [Contact](#)

Part 19: Bringing DataOps to Power BI

17 May 2022 1 min read

Bringing DataOps to Power BI on YouTube

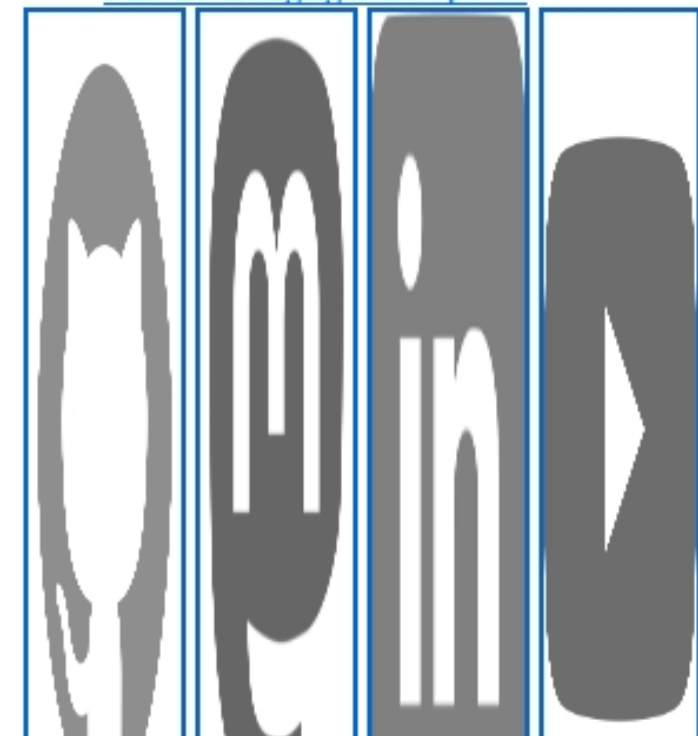
Earlier this year I began a project to put part of this blog series into video format. My aim was to take the principles and best practices described in [Part 3](#), [4](#), and [5](#) of the series and show them in action with a comprehensive demo in Azure DevOps.

I'm pleased to say that video is [now available on YouTube](#). Don't forget to share, like, and subscribe! (yep, I'm a subject to [the algorithm](#) now).

After watching it, let me know what you think on [LinkedIn](#) or [Twitter](#) and now [YouTube](#). I'll continue to be using my blog as my main medium of communication, and based on everyone's feedback I'll plan future videos to encapsulate parts of this series moving forward.

Previous [Part 18: Bringing DataOps t...](#)

Next [Part 20: Bringing DataOps t...](#)



Part 20: Bringing DataOps to Power BI

15 Jun 2022 7 mins read

DataOps Principle #4: It's a team sport

Despite the negativity and seedy underbelly that comes with social media, it does foster an environment for developers and practitioners of technology to collaborate. I spend so much of my time consumed with the work and environments I operate under that I could easily miss the new technologies, techniques and ideas that could spur new opportunities for my team and I.

DataOps Principle #4: Analytic teams will always have a variety of roles, skills, favorite tools, and titles. A diversity of backgrounds and opinions increases innovation and productivity.

Often, we think of a team as the folks who we work with or contract with day-in and day-out. While true, I think of the Power BI community as a part of my team as well. Countless blog articles, forums, videos, and conferences have sparked better ways for my team and I to deliver Power BI solutions. And when I've offered some ideas, the feedback I have received has been just as valuable and has got my wheels turning (and I admit these wheels need a little caffeine and sugar to get moving these days).

A common theme in the feedback has been questions/thoughts on how to bring DataOps to dataflows. Full disclosure, I am new to dataflows as I have had other products available to my team and I for producing curated datasets and dimensions for Power BI (e.g., Data Factory, SSIS), but I can see the immense value dataflows bring to the enterprise by promoting reuse, standards for data, and the convenience in having them as part of the Power BI platform.

So, the first thing to tackle with dataflows was [the same issue we needed to tackle with Power BI Desktop](#), how do I get this low-code's underlying code into source control as seamlessly as possible? If I could do that, all the testing, continuous integration (CI), and continuous deployment (CD) could be easier to implement.

And that was a bit of a challenge because:

- 1) Dataflows are built in the Power BI service; thus, the code sits in the cloud and is not immediately accessible to Git.
- 2) You can [export dataflow code](#) but the steps are manual (done in the browser). You can script the export with PowerShell but how do you know when to run the script? Relying on someone to remember to export the dataflow and push to source control is not really optimal.

With those challenges in mind, I scoured the content out there in the community and started to form a proposed solution. Figure 1 provides an illustration of my first pass.

☒ Proposed Solution

Figure 1 "Proposed solution for automatically storing a dataflow into source control (Git)"

Here are the components of the proposed solution:

- 1) **Azure Data Lake Storage Gen2 (Bring Your Own Data Lake)** - Thanks to a feature that debuted in May 2021, Power BI can store dataflows and the contents (partitions) it builds in your own data lake for an individual workspace. This avoids having to convince your Power BI admin to provide access to a tenant-level data lake and the security considerations that may make the thought of Bring Your Own Data Lake (BYOD) impractical. With BYOD, dataflows and content are stored as shown in Figure 2.

☒ Proposed Solution

Figure 2 "Structure of dataflows in Azure Data Lake Storage Gen2"

The snapshots folder is the key here. Every time a dataflow is created/updated, a version of it is saved into this folder with a timestamp and the latest version is saved in the model.json file located in the parent folder. If we can flag when new entries in the snapshot are created and automatically move the new entry to Git, we can capture every version and avoid a potential race condition on the model.json file. The race condition would involve the time period between this solution receiving notice that model.json has changed and the solution extracting the contents of model.json from the data lake. During that time period model.json may have changed because of another save to the dataflow occurred. Thus, we risk missing a version of the dataflow to save to Git.

- 2) **Event Grid** - This component of Azure allows us to monitor a data lake for new snapshots to appear and issue a notice (topic) in a [publisher/subscription pattern](#). That way Event Grid provides a reliable messaging system to trigger the next component.

- 3) **Azure Functions** - This component has a function triggered by an Event Grid subscription that does the yeoman's work:

1. Grabs the content of the snapshot from the data lake,
2. Issues messages to Azure DevOps to appropriately get the reference id so it can then,
3. push the snapshot file to Git.

When in Git, the files mimic the folder structure in the data lake except that each snapshot is named model.json so we have a history of changes in Git. This will make the work of pipelines and testing easier to work with as well.

- 4) **Azure Key Vault** - For the Azure Function to interact with Azure DevOps it needs a Personal Access Token with read, write, manage permissions to the code in Git. Microsoft recommends using Key Vault instead of Application Settings to store this sensitive information.

- 5) **Managed Identity** - This is the glue that holds a lot of this solution together. The Azure Function needs access to the data lake and the key vault and by making the Azure Function a managed identity you can grant access to each component without the need for maintaining a new set of passwords (and we all have too many passwords to maintain these days).

Sharing The Proposed Solution

In the spirit of [make it reproducible](#) I have scripted the installation on this proposed solution and [made it available on GitHub](#). Let me know what you think on [LinkedIn](#) or [Twitter](#).

From my own reflection there are some things that I think could make the solution better:

- 1) **Durable Functions** - I tried to make the proposed solution use the consumption license model to keep costs down, but a [Durable Function](#) could make the Azure Function more responsive to changes and allow for [function chaining](#) which could make the custom code more manageable (and easier to test).

- 2) **Easier setup for between the workspace and data lake connection** - During the course of setting this solution up, the "Access Denied" message from connecting a Power BI workspace to the data lake befuddled me. With the same permissions on the data lake it could take between 1 and 3 tries (without any manual permission changes) before the connection was established. I even [put in a pull request](#) in the hopes that the instructions could be better. I would like to see this installation step more automated and reliable.

- 3) **Automated PAT token generation** - This capability is available but [is still in preview](#). I would prefer to automate this installation step when it becomes generally available or figure out a better way to extend the Function's managed identity permission to Azure DevOps and get rid of the PAT Token altogether.

What do you think could make it better? Any thoughts on making this scalable?

A team effort

I would not have accomplished anything workable or anything so quickly without the tech. communities' efforts to enlighten and inspire. So, it is time to say thanks to the team:

Matthew Roche for the [series on dataflows](#) (a must read and watch)

Olivier Miossec for making it easier for me to get [PowerShell to work with Azure DevOps](#).

Nicola Suter for guidance on integrating [PowerShell modules in Azure Functions](#).

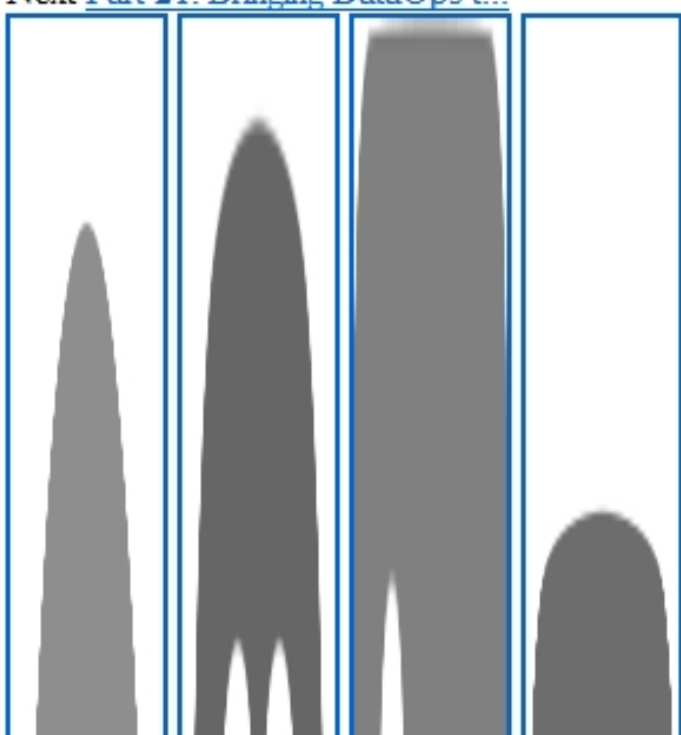
Wojciech Lepczyński for teaching me how to work with [Azure Functions using the Azure CLI](#).

And those who keep the Microsoft documentation up-to-date with articles on [ADLS Gen2 and Power BI](#) and tutorials on using [Azure Event Grid](#).

In my next article we'll continue Bringing DataOps to dataflows, stay tuned team.

Previous [Part 19: Bringing DataOps t...](#)

Next [Part 21: Bringing DataOps t...](#)



Part 21: Bringing DataOps to Power BI

12 Jul 2022 6 mins read

DataOps Principle #15: Quality is Paramount

In my [last entry](#) I explained how to update a dataflow in the Power BI Service and have it automatically check-in those changes to source control. Now that we have the code in Git, what's next?

Here is where we turn to the principle of "Quality is Paramount":

*DataOps Principle #15 - Analytic pipelines should be built with a foundation capable of **automated detection of abnormalities and security issues in code, configuration, and data, and should provide continuous feedback to operators for error avoidance.***

Dataflows are the epitome of an analytic pipeline, and therefore we must create a foundation for **testing** for issues with the code and the data in an automated fashion.

But we are challenged by:

1. The lack of a built-in testing tool within Power BI. In [Part 4](#) we used DAX Studio as an external tool to test the datasets. However, that is not an option with dataflows because you cannot query the output of a dataflow with DAX.
2. The lack of an easily accessible testing pattern within Power BI for data analysts. As a self-service business intelligence tool, Power BI's largest user base are data analysts, not developers/computer scientists who are familiar with writing code that tests code. Case in point, when you pass the [PL-300](#) exam, you become a certified Power BI *Data Analyst* and not a certified Power BI Developer.

Looking For A Solution

This led me to look around for some options to mitigate these challenges. I came across [an article written at endjin](#) by James Broome. In it he explained that the team at endjin used SpecFlow to automate testing. SpecFlow implements a behavior driven development approach that uses the Gherkin Language to define tests through written narratives and acceptance criteria (see example below).

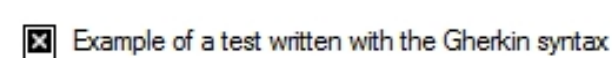
 Example of a test written with the Gherkin syntax

Figure 1 "Example of a test written with the Gherkin Syntax courtesy of [Wikipedia](#)

Within SpecFlow, tests are written in .NET and executed using .NET. Sure, I love .NET, but I was looking for a way to execute the tests so the data analyst could build/run the tests in Visual Code on their computer with very little coding; furthermore, they could also run the same tests in Azure DevOps or GitHub Actions to automate testing when dataflows are updated.

This led me to PowerShell (yes, I know it runs on .NET), and [version 4 of Pester](#) which is a test runner that implements the Gherkin Language.

A Potential Solution

With version 4 of Pester, the sentences written in the Gherkin syntax (which you can think of as plain English) are a "feature" file and look something like this:

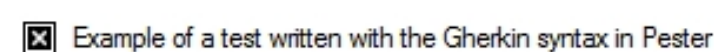
 Example of a test written with the Gherkin syntax in Pester

Figure 2 "Example of a test written with the Gherkin Syntax in Pester

Each sentence can be backed by a "steps.ps1" file written in PowerShell where a function executes a sentence based on its intent. Therefore, starting from line 4 in Figure 2 you could have a set of corresponding PowerShell functions that:

1. Lines 4-6 - Verify a connection to the Azure Data Lake Storage can be made, and you have access to the storage account and model.json file.
2. Lines 9-15 - Inspect the contents of the model.json file in source code, verify it has an entity (table) and that its schema contains a predefined set of columns and format.
3. Lines 17-20 - Reach out to the Azure Data Lake Storage and verify the number of records ingested.

This method is quite powerful in that if you construct the "steps.ps1" files correctly, many of the tests a data analyst could write in plain language ("feature" file) could be executed with little need to look at the PowerShell code written in the "steps.ps1" file.

Running the Tests

So now you may be thinking, alright this Pester/Gherkin thing could be quite useful, but what is required to run these tests? Well, if you already bought into [Part 20](#), you have VS Code (free), and you've got the project cloned to your local machine with Git (free), then running the tests just involves running "Invoke-Gherkin" in the terminal. It will run through all the tests in your project and tell you what has passed or failed.

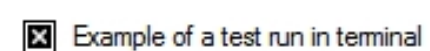
 Example of a test run in terminal

Figure 3 "Example of tests executed in VS Code terminal with Pester

Oh, and even better, these tests can automatically run in Azure DevOps with the same "Invoke-Gherkin" statement. I'll save that for an upcoming post as it involves some level of orchestration (hint, hint) to explain.

Best Practices

For those looking to test their dataflows, I have some best practices that should help mitigate errors going to production.

1. Have a Test Dataset "In order to test effectively you need to have a known underlying dataset to validate your Power BI dataflow. For example, if you import a SQL table, adjust the query to a specific period of time. The goal is to have a static set of data to work with, so the only variables that would change during a test is the code you or your team has changed in the dataflow.
2. Leverage Parameters in your Dataflow "Eventually when you want to transition from the test dataset to a staging or production environment you don't want to have to manually update the code. Parameters can allow you to manage connection details in the Power BI Service between your environments (dev/test/prod).
3. Test the Schema "For each table (entity) produced by a dataflow make sure the columns match the names and formats expected. Unexpected schema changes are a common cause for broken analytic pipelines, so this type of test is a must for me.
4. Test the Content "Columns in each table produced by a dataflow should meet some expectation you can test, such as:
 - o The uniqueness of values in a column (no duplicates).
 - o The min and max values for an numeric column.
 - o A column of strings match a regular expression.
 - o A column has no blank values.

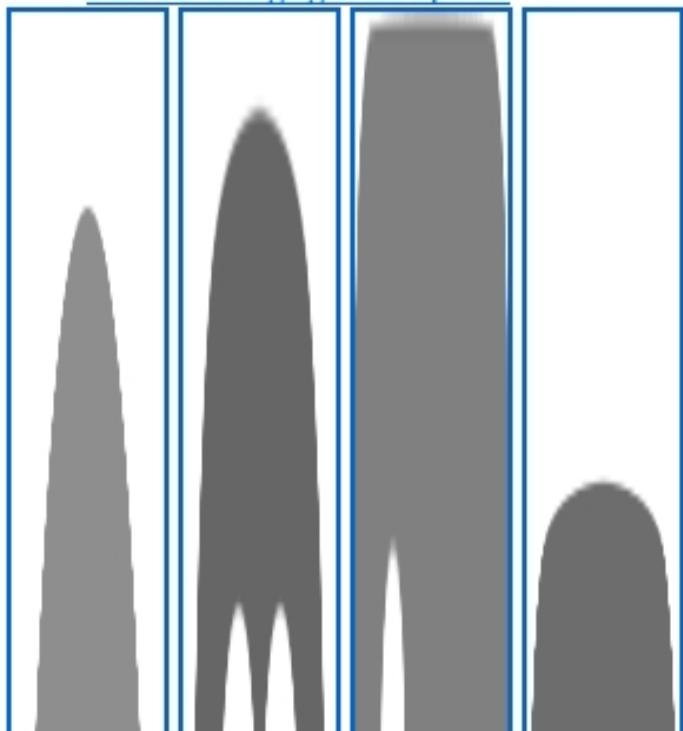
Sharing The Proposed Solution

With the Proposed Solution and Best Practices in mind, I've shared instructions on how to run these type of tests with a sample dataflow [on GitHub](#). My hope is that you can leverage these samples to start to build a suite of dataflows backed by source control, and tested using a framework which data analysts can use.

Let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 20: Bringing DataOps t...](#)

Next [Part 22: Bringing DataOps t...](#)



Part 22: Bringing DataOps to Power BI

06 Sep 2022 6 mins read

DataOps Principle #10: Orchestration

In my last two entries I explained a way to integrate Power BI dataflows into source control and then write test cases using the Gherkin-language. If you’ve read some of the prior entries, you may probably know where this article is going| orchestration.

DataOps Principle #10 - The beginning-to-end orchestration of data, tools, code, environments, and the analytic teams work is a key driver of analytic success.

Orchestration requires a level of automation, so with that in mind, I focused on automating three parts:

- 1) Figure out a way to manage a roll-back to a prior version of the dataflow in source control so the Power BI service could be updated automatically.
- 2) Automate the testing and publishing of test results.
- 3) Script-out the configuration to make it easy to setup new projects in the future.

The Challenges

Figure out a way to manage a roll-back to a prior version of the dataflow in source control so the Power BI service could be updated automatically

I didn’t anticipate #1 to be so difficult. I was faced with several challenges I’d like to share:

- 1) There is no easy way to discern through git history if a new push (an update to the repository in Azure DevOps) is actually a roll-back. I scoured several Stack Overflow threads and there is quite a history of colleagues trying to do the same thing. The most straightforward answer I could come up with was to start the commit message with “Revert:” and use a regex (always love a chance to use regex) to handle this situation.
- 2) As of August 2022, dataflows are owned by one user, and back when dataflows were made [generally available](#), Microsoft provided the ability to take over ownership in the interface. So, my plan was to take over the dataflow with my service account running in Azure DevOps and replace it with the revert version. I made sure my service account had admin rights to the workspace with the dataflow and then looked for the API endpoint to take over ownership in a dataflow. I knew one existed for taking over a dataset, so in 3 years surely there would be one for taking over ownership in a dataflow. **I was wrong**. I resorted to network snooping with Developer Tools on the browser and found Microsoft uses an internal API call to do it: `https://wabi-us-east2-b-primary-redirect.analysis.windows.net/metadata/dataflows/xxxxx-xxxx-xxxx-xxxx-xxxxxxxxx/takeover`. It does exist, but is not available to the public (sigh). I know I’m not the only one with this issue| please up-vote this suggestion on [Power BI ideas](#).

Example from Microsoft on taking over a dataflow in the Power BI Service

Figure 1 “ Example from Microsoft on taking over a dataflow in the Power BI Service. But no API endpoint to do the same ”

- 3) Then I thought I’d be slick and use the Bring-Your-Own-Data-Lake I needed for the source control piece and just update the model.json there. Well, no luck there either. You can update the model.json but those changes are not reflected in the service or, if they are, do not immediately update the service which would be problematic.

I left feeling a little defeated and decided to address this later when either the API endpoint is made available, or some other suggestions come up that spark a solution.

Automate the testing and publishing of test results.

At this point I was hoping for an easy win to lift my spirits. The challenge I needed to overcome was allowing a service account to run the tests AND have access to the data lake in order to run tests on the partitions. Thankfully Azure DevOps offers that ability with an [Azure Resource Manager Service Connection](#). This creates an app service principal I can use to grant read access to the data lake. Then with the help of an Azure PowerShell script task, I can run the Invoke-Gherkin command (I explained this command in [Part 21](#) to run the tests. Thankfully, the next step to publish the test results was even simpler. Since the Invoke-Gherkin command produces test results in the [NUnit](#) format, we can use the built in “Publish Test Results” step and load the results into our test results library.

Example of loaded test results in Azure DevOps

Figure 2 “ Example of loaded test results in Azure DevOps

Script-out the configuration to make it easy to setup new projects in the future.

Alright, so I anticipated this objective was going to be a challenge. From prior projects I had built scripts to automatically setup an Azure DevOps project and pipeline, but never before had I created the Azure Resource Manager Service Connection with the Azure CLI. At first, I read the Microsoft documentation which offered ambiguous guidance on using the Azure CLI. I then turned to blogs written by colleagues who may have posted on the topic. After reading articles by [Rodney Almeida](#) and [Sander Rozemuller](#) I was ready to experiment. After a few rounds of trial-and-error I figured out that I needed to use the Azure CLI to create an app service principal first, make sure it had explicit read rights to the Azure Data Lake and then setup the Azure DevOps Service Connection using the service principal I created. There may be a better way to do this, but I got consistent results using this script, so that was good for now.

The incantation I needed to automate the creation of an Azure Resource Manager Service Connection

Figure 3 “ The incantation I needed to automate the creation of an Azure Resource Manager Service Connection

Orchestration

Well, completing 2 out of 3 automations isn’t bad, so here is a visual of the orchestration in action.

I’ve highlighted the new parts to this process in the image below and here’s what they do:

- 1) The Azure Pipeline sees an update to the repo has occurred and starts the build.
- 2) The Azure Resource Manager Service Connection is used to run the task that calls Invoke-Gherkin.
- 3) All test cases in the project are ran and the results are stored in the Test Plans component of Azure DevOps for review.

New automations highlighted

Figure 4 “ New automations highlighted

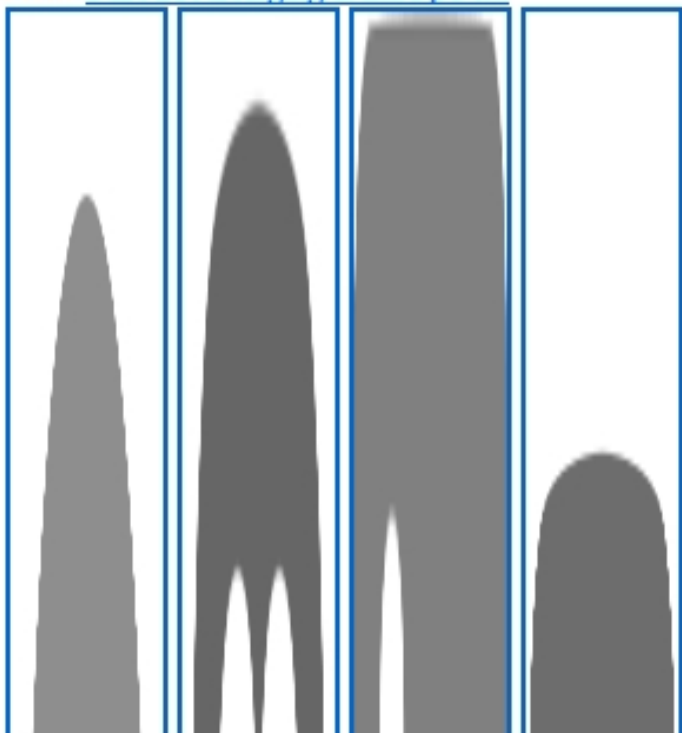
Sharing The Proposed Solution

I’ve shared my code and instructions on how to setup and orchestrate a build with a sample dataflow on [GitHub](#). My hope is that you can leverage these samples to start to build a suite of dataflows backed by source control, tested using an easy-to-use framework, and automated through orchestration.

Let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 21: Bringing DataOps t...](#)

Next [Part 23: Bringing DataOps t...](#)



Part 23: Bringing DataOps to Power BI

04 Oct 2022 11 mins read

DataOps Principle #14: Analytics is Manufacturing

Let's switch gears. We'll take a break from talking about dataflows and turn attention back to Power BI datasets. In [Part 22](#), I proposed using the Gherkin language to implement testing - thanks to version 4 of Pester. At the time I was using it for dataflows, but I have since realized I could use it to gain efficiencies while testing Power BI datasets.

DataOps Principle #14: Analytic pipelines are analogous to lean manufacturing lines. We believe a fundamental concept of DataOps is a focus on process-thinking aimed at achieving continuous efficiencies in the manufacture of analytic insight.

Areas of Improvement

After spending some time with testing Power BI datasets using a technique from [Part 4](#), I identified some areas for improvement:

- Multiple DAX test files - As the number of DAX test files grew, data analysts had to have more test files open in something like [DAX Studio](#), then go through each file and run the query manually to make sure the tests passed in each file. There had to be a more efficient way.
- Integrating different types of tests - While DAX-based tests helped test certain aspects of the datasets (like DAX measures, calculated columns, and some table contents), I wanted to merge other ways to test the datasets. This includes [testing the schema](#) and checking column values against [regular expressions](#) (you know I love regular expressions). If the tests could be integrated, then I could possibly track test results better to support [monitoring for quality and performance](#).
- Simplifying the testing process - As I have stated before in [Part 21](#), Power BI's largest user base is data analysts and not developers/computer scientists who are familiar with writing code that tests code. I wanted to make it as simple as possible to both create the tests, and run the tests locally. I also wanted to make sure the same method used to test locally could be leveraged in the pipeline during [orchestration](#).

Making the Improvements

[Pester Version 4 and its Gherkin language implementation](#) seemed like a logical approach because I could abstract the technical work within the `steps.ps1` file from the `feature` file which would contain tests that could be written in plain language by a data analyst. But there are a few challenges when running the same tests both on your local machine or against the Power BI Service (via a Continuous Integration pipeline):

- Challenge #1 - Are we running these tests locally? When running tests locally we need to connect to the Power BI dataset differently than when the Power BI dataset is in the service. If local, Power BI creates a network port to connect to via the localhost. In the service, the connection requires an XMLA connection with a different syntax.
- Challenge #2 - Which Power BI files are opened locally? When running tests locally we need to find which network port corresponds to each Power BI file that may be opened. We also need to make sure we run tests for only the opened Power BI files.

After some trial-and-error, I produced a solution thanks to (1) how the awesome gentleman at [GuyInACube](#) used PowerShell to identify the local network ports opened for Power BI Desktop files, and (2) by reviewing DAX Studio's way to figure out which network port was associated with the Power BI file opened. This is a great case for why open source is awesome, and I'll return the favor by making the [Get-PowerBIREportsOpenedLocally](#) module open source as well.

With those challenges overcome, I could build a potential framework to the three areas of improvement. Based on [a sample project I have uploaded to GitHub](#) here is what it looks like:

The Testing Structure

Building and running tests are based on the Behavior Drive Development (BDD) concept. Pester Version 4's implementation of the Gherkin language facilitates BDD testing by defining tests through written narratives and acceptance criteria.

Building tests are based on two files:

.feature file

Within this project there are two sample `feature` files:

SampleModelSchema.feature

This feature file shows how you can test the schema and content of the Power BI dataset. The following is an example implementation:

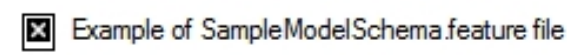
 Example of SampleModelSchema.feature file

Figure 1 - Example of SampleModelSchema.feature file

Each sentence executes a test, which for the example above does the following:

- Line 1: Defines the type of test you are performing.
- Lines 2-12: The Background section verifies the Power BI report exists, you have access to the files (in this case the wonderful Tabular Editor) to get the schema and verify you can retrieve it.
- Lines 14-20: This Scenario is a set of steps to verify the `AlignmentDim` table exists and the schema defined is correct.
- Lines 22-25: This Scenario is a set of steps that verifies the `AlignmentDim` table has certain columns that match the regular expressions defined.

SampleModelDAX.feature

This feature file shows how you can test a Power BI dataset using DAX. All tests should result in an output that is consistent so that the automated tests can verify the tests pass. Fortunately, with DAX, we can create that schema. The example below provides the output of a set of test cases and follows a simple format:

 Example of DAX testing schema

Figure 2 - Example of DAX testing schema

- Test Name- Description of the test prefixed by MEASURE, CALCULATED COLUMN, or TABLE to indicate what part of Power BI model is being tested.
- Expected Value - What the test should result in. This should be a hardcoded value or function evaluated to a Boolean.
- Actual Value - The result of the test under the current dataset.
- Test Passed - True if the expected value matches the actual value. Otherwise, the result is false.

By leveraging a consistent schema to build tests using DAX, we can automate the testing of multiple DAX files. In this template, we have 3 files (CalculatedColumnsTests, MeasureTests, and TableTests). Therefore, the feature file to run multiple DAX test files is shown in the example below.

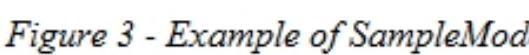
 Example of SampleModelDAX.feature file

Figure 3 - Example of SampleModelDAX.feature file

Each sentence executes a test, which for the example above does the following:

- Line 1: Defines the type of test you are performing.
- Lines 4-5: Verifies that Power BI reports exists.
- Lines 7-9: Using the Scenario Outline we can create a loop to run each DAX test file.
- Lines 11-14: With the Scenario Outline defined we can just add the name of the DAX file to the table containing one column called `TestFile`. Then each file will be run (DAX query) against the Power BI dataset and the test will verify the expected values and actual values match.

.steps.ps1 file

Each sentence in the feature file is backed by a `steps.ps1` with the same name as the feature file.

Since we want to take advantage of the same PowerShell code to run similar schema and DAX tests, all `steps.ps1` files reference the file `Test-Support.steps.ps1`.

The `Test-Support.steps.ps1` file supports the following test cases:

Background Tests

Given that we have access to the Power BI Report named `{PBIFile}`

Verifies the parameter `{PBIFile}` exists as a `.pbix` file within a subfolder under the `/Pbi` folder.

Schema Tests

And we have the following properties

Accepts a json file that lists the location of Tabular Editor and the `GetSchema.cs` file. This merely checks to see if those files exist so we can conduct further tests.

And we have the schema for `{TableName}`

Accepts the `{TableName}` parameter and uses `TabularEditor` and the `GetSchema.cs` file to pull the schema information for the table. If the test case can get this information, it passes.

Then it should `{Contain or Match}` the schema defined as follows:

After the [prior schema test](#) this test accepts a table of information with the columns Name, Type, and Format such as:

Alignment	string	Name	Type	Format

- Name: This is the name of the column.
- Type: This is the type of the column (e.g., string, number, boolean).
- Format: This is the format of the column. You can leave this blank if the format does not need to be tested.

This test accepts a parameter `{Contain or Match}`. If the parameter entered is `Contain` then this test will make sure each column exists and matches the type and format. If the parameter entered is `Match` then this test will make sure the table has all the columns defined in the test, that each column exists, and that each column matches the type and format. The `Match` value is strict and makes sure no new columns exist in the dataset compared to the defined table in the feature file.

Regex Tests

Given we have a table called `{TableName}`

Accepts the `{TableName}` parameter and makes sure table exists in the dataset.

And the values of `{ColumnName}` matches this regex: `{Regex}`

After the [prior table test](#), this function accepts the `{ColumnName}` parameter and `{Regex}` parameter. This verifies that the column in the table passes the regular expression. The Regular Expression format follows the [.Net Regular Expressions format](#).

DAX Tests

Given we have the `{TestFile}` file

Verifies the parameter `{TestFile}` exists as a `.msdax` or `.dax` file within the subfolder defined [by the PBIFile in the Background tests](#).

Then `{TestFile}` file should pass its tests

This queries the dataset defined [by the PBIFile in the Background tests](#) using the DAX file defined by the parameter `{TestFile}`. If the query returns an [acceptable schema](#) and all the expected values equal the actual values in the query results, the test will pass.

Running Tests

This project has a script called `Run-PBITests.ps1` that exists at the root of the project.

This script allows you to run the tests you created for each Power BI file that is open. Here are the steps:

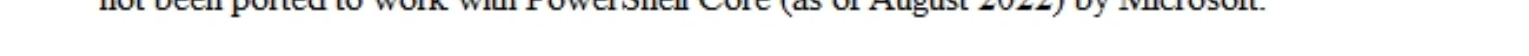
- Within Visual Studio Code, open your project folder.
- Then within Visual Studio Code click the terminal menu option and select "New Terminal".


- From the terminal enter the command "Powershell -NoExit" This commands makes sure we are running classic PowerShell and not PowerShell Core. The command `Invoke-AsCmd` has not been ported to work with PowerShell Core (as of August 2022) by Microsoft.


- Run PBI Tests


- Then from the terminal enter the command `./Run-PBITests.ps1`

- If the test cases pass, then you will see in the terminal a confirmation of success with a message "SUCCESS: All test cases passed."


- If a test fails, then you will see in the terminal which test cases failed (see example).


Orchestration

One of my favorite parts about Pester is that it will allow you to produce the test results in the NUnit format. Therefore, now when the testing completes in a CI pipeline it can publish those results to Azure Test Plans. Thus, we have a more centralized way to monitor and track testing.

 Example of Publish Test Results in Azure DevOps

Figure 4- Example of Publish Test Results in Azure DevOps

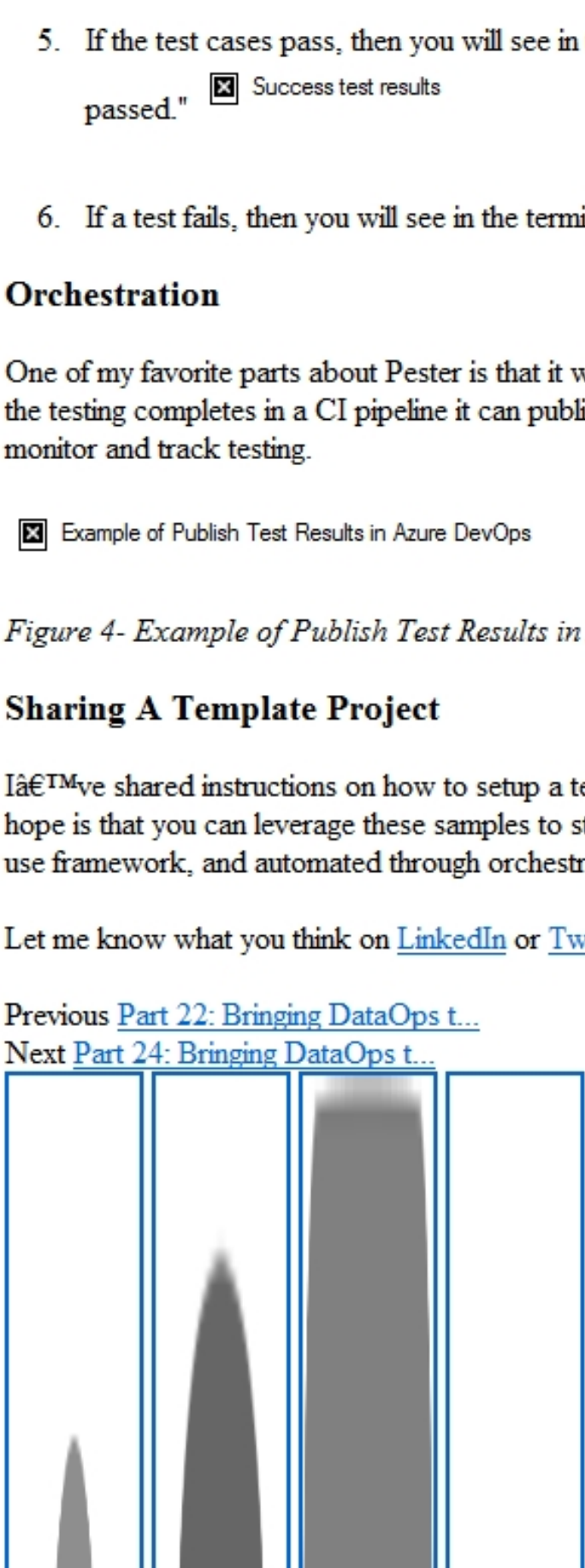
Sharing A Template Project

I've shared instructions on how to setup a template project, cloning that project locally, and running tests on [GitHub](#). My hope is that you can leverage these samples to start to build a suite of datasets backed by source control, tested using an easy-to-use framework, and automated through orchestration.

Let me know what you think on [LinkedIn](#) or [Twitter](#).

Previous [Part 22: Bringing DataOps t...](#)

Next [Part 24: Bringing DataOps t...](#)



Part 24: Bringing DataOps to Power BI

02 Nov 2022 6 mins read

DataOps Principle #16 - Monitor Quality and Performance

Over the past year I've been leveraging [Application Insights](#) on my blog to monitor traffic and issues. Surprisingly, [Part 2](#) continues to get regular traffic and the template I provided to help monitor Power BI refreshes receives regular traffic as well. While it's exciting that people are interested in making sure their Power BI datasets refreshes are healthy and thus embracing "Monitoring for Quality and Performance" within DataOps, I knew the template I offered could be better.

And why is that? Well, I'm glad you asked. Here are two reasons why:

1) **Complicated Setup** – The template I offered required you to setup an application registration and this always presents a risk for complications. It's been my experience that making sure the application grants admin consent to APIs or asking an Azure administrator to grant admin consent, is a common oversight. In addition, making sure the Resource URL is the correct URL could also be problematic. Power BI Commercial, Government Community Cloud (GCC), GCC-High, and other sovereign clouds all have different Resource URLs. Thus, I wanted to streamline the process to reduce issues during setup.

2) **Listening to an Expert** – The template was based on querying [Power BI's REST APIs](#) using a OAuth2 token generated within the dataset. This requires you to save a password within the dataset. Now, if you protect access to the dataset, you shouldn't be worried; however, I knew that this wasn't the best option. I was further pushed to find a better template when a [blog post was published by Chris Webb](#) a month after I published [Part 2](#). In his post it summarized Microsoft's advisement that if you want to connect from Power BI to a REST API that uses OAuth2 authentication (and that's what I was doing in the template), then you need to build a custom connector.

This may not be a surprise to you, but there are custom connectors out there that you can pay for, and it enables you to query Power BI's REST APIs. Kudos to those companies seeing a good business opportunity, but I always felt that you shouldn't have to pay for a connector that queries Power BI **within** Power BI. So, I went looking for ways to build an open-source connector.

Building a Custom Connector

Thanks to [Microsoft's training](#), I was able to understand how to build a connector. Yet, I needed Visual Studio to build a connector, and if I wanted to easily enable others to contribute to this connector, I needed to use Visual Studio Code. Thankfully I noticed in [September 2022](#) that a SDK was being developed by Microsoft to offer that capability in Visual Studio Code. I downloaded the extension and began to build the connector.

After some trial and error, I was able to get a version built. For this connector I'd like to share some of my design considerations:

1) **A Catalog of Functions** – I wanted those using this connector to feel empowered to innovate, so I wanted to make this connector a gateway to many of the read-only endpoints within the Power BI REST APIs. Therefore, I borrowed Kim Burgess's design pattern I wrote about in [Part 13](#) and, thus, each endpoint is its own Power Query function.

Figure 1

Figure 1 – A Catalog of Power Query Functions for Power BI REST APIs

2) **Raw JSON** – Within the connector I need to make a critical design, when someone calls a Power Query function that hits a Power BI REST endpoint, does that function return a table transforming the JSON data returned from Power BI **OR** do I just return the raw JSON data and let the Power BI data analyst transform the JSON? I ended up choosing the latter option because I've worked in different sovereign clouds and I knew from experience that the Power BI REST APIs and the JSON properties returned by those endpoints differed across the cloud environments. If I made that decision, I would be on the hook to keep track of the changing JSON schemas as Microsoft updates them AND keep track of the differences in those schemas between the cloud environments. I knew this wasn't feasible given the pace of change from Microsoft and the unpredictable change of bureaucratic institutions. This is a long-winded way of saying the user of this connector will need to use the transform JSON step in Power Query, choose what properties they need, and not have to wait on this custom connector to be updated when a new property appears in the API (you're welcome! hopefully).

Figure 2

Figure 2 - Raw output from GetGroups so you can choose what JSON properties you need

Since the Resource URLs and API endpoints are different between commercial and sovereign clouds, I did have to make different customer connectors. Not a great option (and I'm open to ideas if this can be consolidated), but it works. As of November 2022, I have published three connectors:

- 1) [Power Query Custom Data Connector for Power BI REST APIs \(Commercial\)](#)
- 2) [Power Query Custom Data Connector for Power BI REST APIs \(Government Community Cloud\)](#)
- 3) [Power Query Custom Data Connector for Power BI REST APIs \(Government Community Cloud High\)](#)

Sharing The New Template

With a new custom connector, the setup for running on your desktop and the instructions to run in a gateway (dataset refreshes with custom connectors require a gateway, sorry) are simplified. Plus, no password in your Power BI file! whew. I'm happy to [share a new template on GitHub](#) that uses this connector.

Figure 3

Figure 3 - A new Power BI template to monitor dataset refreshes using the custom connector

This updated template includes:

- 1) The ability to pull data refresh history and schedules across multiple workspaces.
- 2) The ability to see the refresh schedules for the next seven days. This is very helpful when planning for changes or reviewing maintenance windows.

Figure 4

Figure 4 - Example of the Upcoming Refreshes tab in the template

3) The ability to choose your time zone. The Power BI services lets dataset owners choose which time zone to schedule their refresh times. This can lead to problems interpreting refresh times since the Power BI API returns the schedule for the time zone and not in UTC. This template will convert the time zones to UTC and then let you decide which time zone you want to see in the report. It also supports US Daylights Savings, and I'm open to incorporating other daylight savings rules (like the European Union) but I would need help testing.

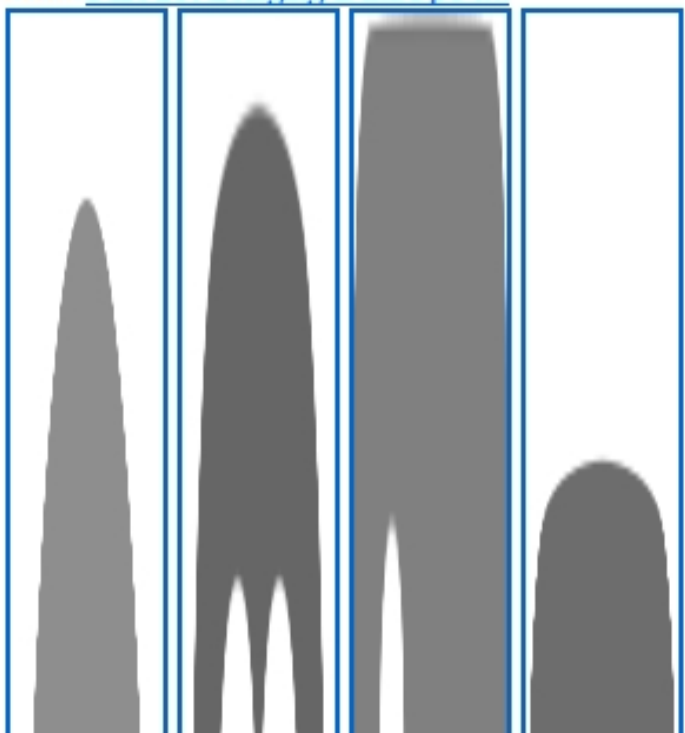
Feedback

If you're interested in testing or contributing to a version of this custom connector, please reach out to me on [LinkedIn](#) or [Twitter](#). If you have feedback on the [new template for monitoring Power BI Refreshes](#), please contact me as well.

Finally, if you're going to be at the [PASS Data Community Summit](#), November 15th-18th 2022, I hope to see you there and see you at my session on Wednesday, November 16th at 4:15 p.m. local time.

Previous [Part 23: Bringing DataOps t...](#)

Next [Part 25: Bringing DataOps t...](#)



Part 25: Bringing DataOps to Power BI

09 Jan 2023 13 mins read

DataOps Principle #18 – Improve Cycle Times

Late in September 2022, I had the chance to meet with [Mathias Thierbach](#), author of a phenomenal tool called [pbi-tools](#). During that time, we had the chance to talk about a number of DataOps related topics including version control and testing. One area that we had a good time discussing was how we could test the visuals and the look-and-feel of a Power BI report. If we could automate some visual testing, Power BI developers could improve the cycle time to deliver high quality solutions.

DataOps Principle #18: We should strive to minimize the time and effort to turn a customer need into an analytic idea, create it in development, release it as a repeatable production process, and finally refactor and reuse that product.

Since September we’ve been quite busy with our day jobs, but recently I had to improve the consistency in the user experience and accessibility standards for a project with Power BI reports. For the project team, the effort to manually check for consistent branding, user experience, accessibility, etc. was a manual endeavor, it impeded delivery timelines, and led to the deployment of solutions that did not completely meet standards. While Microsoft provides guidance for [creating accessible reports](#), automating checks for this guidance, and thus improving cycle times, posed one big challenge:

1) Access to visual and report settings – Through Dynamic Management Views and XMLA, Microsoft offers ways to [interface technically with](#) the data model, but *an out-of-the-box way to interface technically with the visual or report settings does not exist.*

Thankfully, pbi-tools offered a third-party solution. With its [extract](#) command, the tool can pull all the settings (data model and visuals) from a .pbix file into a series of folders and file formats that make the visual and reports settings much more accessible. Furthermore, with its watch command, the tool will automatically run the extract command every time you save the file. With pbi-tools as a solution, now I could focus on the steps towards improving cycle time.

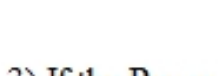
The First Step

While I had become familiar with pbi-tools, the first step was to simplify the watch and extract process for other Power BI analysts. Asking analysts to figure out the network port number that their current Power BI file was using and making sure they extract the contents of the Power BI file to the right folder was prone to error. So, for our [project template](#), I added a simple PowerShell command called Watch-PBI.ps1.

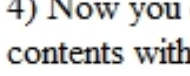
Watch-PBI.ps1

When updating Power BI files within this project, you can track changes by following the steps below.

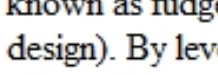
1) Within Visual Studio Code click the terminal menu option and select “New Terminal”.



2) From the terminal enter `.\Watch-PBI.ps1 -FileName {FileName}` where {FileName} is the name of the Power BI file in double quotes (example below).



3) If the Power BI file is not opened already, it will launch Power BI Desktop and initiate the extract and watch. If the Power BI file is already opened it will immediately initiate the extract and watch.



4) Now you can update the Power BI file and when you save the file locally (Ctrl+S), pbi-tools will automatically update the contents within the PbiProj folder identifying the changes.

The Second Step

Next it was important to enable a Power BI analyst to generate the tests from a template. By following a template approach, I could empower the Power BI analyst to adjust any of the tests based on requirements or make exceptions to the rules, also known as fudge-ability (this term I learned from *The Inmates Are Running the Asylum*, an excellent book on user interface design). By leveraging the Gherkin approach I’ve discussed in [Part 23](#), I built another PowerShell command called Generate-Tests.ps1

Generate-Tests.ps1

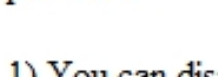
Generating tests are based on the Behavior Drive Development (BDD) concept. Pester Version 4’s implementation of the Gherkin language facilitates BDD testing by defining tests through written narratives and acceptance criteria.

Tests are based on two files:

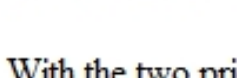
- .feature file - The type of tests written in plain language.
- .steps.ps1 - The implementation of the tests.

To generate a template, here are the steps:

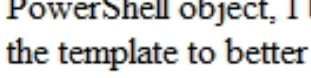
1) Within Visual Studio Code click the terminal menu option and select “New Terminal”.



2) Then from the terminal enter the command `.\Generate-Tests.ps1 -Filename {PBIX} -Type Visuals` where {PBIX} is replaced with the name of the Power BI file you wish to generate test cases for visuals.



3) If the command runs successfully, then two files will appear (example outlined in orange below) in the project’s sub-folder.



Visuals Test Contents

Within the Visual.feature file are several tests that are described in the next section; however, there are some important things to point out:

- 1) You can disable and/or skip a test by putting a hash tag (`#`) preceding the test.
- 2) You can disable running a whole scenario or scenario outline by putting a hash tag (`#`) preceding the appropriate scenario or scenario outline.

The Final Step

With the two prior steps completed, we had a good foundation to leverage pbi-tools and Gherkin in a repeatable manner for testing visuals and report settings. Now I just needed to break out some more PowerShell to (1) extract the contents into a PowerShell object that would represent the visuals and report settings and (2) define tests against that PowerShell object.

I built a PowerShell module in the project template called [Get-PBIToolsConfigurations.psm1](#) to extract the contents and implemented my own unit tests using Pester (testing code that tests other code, I just can’t help myself). Then with a PowerShell object, I built a preliminary set of tests. This next section is quite lengthy, but I go into each of the 22 tests available in the template to better explain the level of automated testing available (or you can [skip to the end](#), I won’t judge).

Visual Tests for Reports

1) Given that we have the report settings

This verifies that a PbiProj folder exists for the report and has extracted data using pbi-tools.

2) Then the default section is the {1st or 2nd or 3rd} section

This verifies that the nth tab (typically 1st) is the default tab that opens in the Power BI service for the report. Often when working on a report it’s easy to save and publish on a tab that is not considered the default. This helps verify that’s not an issue.

3) Then all report-level measures have a prefix: {MEASURE_PREFIX}

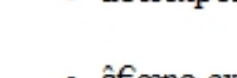
For measures created on reports or reports that are connected to a Power BI dataset, you can implement a prefix naming convention using the parameter {MEASURE_PREFIX}.

4) And the report uses a custom theme named {THEME_FILE}

Accepts {THEME_FILE} parameter and will verify the report uses that theme file.

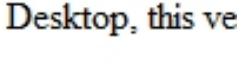
5) And the Persistent Filters setting is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below.



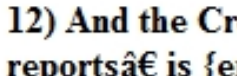
6) And the Visual Option {Hide the visual header in reading view} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below (outlined in orange).



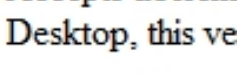
7) And the Visual Option {Use the modern visual header with updated styling options} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below (outlined in orange).



8) And the Visual Option {Change default visual interaction from cross highlighting to cross filtering} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below (outlined in orange).

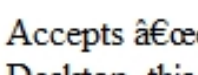


9) And the Export data setting is {export summarized data only, export summarized and underlying data or no export allowed}

Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below.

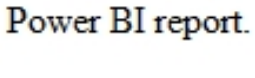
Accepts:

- {export summarized data only} - option 1 in image below.
- {export summarized and underlying data} - option 2 in image below.
- {no export allowed} - option 3 in image below.



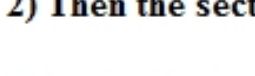
10) And the Filtering experience {Allow users to change filter types} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below (outlined in orange).



11) And the Filtering experience {Enable search for the filter pane} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below (outlined in orange).



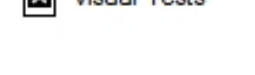
12) And the Cross-report drillthrough setting {Allow visuals in this report to use drillthrough targets from other reports} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below.



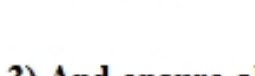
13) And the Personalize visuals setting {Allow report readers to personalize visuals to suit their needs} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below.



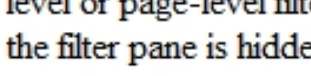
14) And the Developer Mode setting {Turn on developer mode for custom visuals for this session} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below.



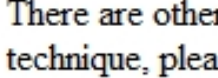
15) And the Default summarizations setting {For aggregated fields, always show the default summarization type} is {enabled or disabled}

Accepts {enabled} or {disabled} parameter. Under the Options->Current File->Report Settings file in Power BI Desktop, this verifies the setting’s status in the image below.



Visual Tests for Sections

These tests rely on a scenario outline so it can run tests for each section (example below). Sections are considered tabs in the Power BI report.



1) Given that we have the section: <Section>

This verifies the section exists in the report configuration files.

2) Then the section has their {canvas or wallpaper} set with the background image named {IMAGE_FILE}

This accepts the canvas or wallpaper parameter; the settings are found in the visualizations pane for a specific section (outlined in orange in the image below)



This also accepts the {IMAGE_FILE} parameter which should be the name of the image used in with the wallpaper or canvas.

3) And the section has a width of {WIDTH}px and a height of {HEIGHT}px

This accepts the {WIDTH} and {HEIGHT} parameters in pixels and verifies the Canvas Settings height and width (image below).

Visual Tests for Visuals

These tests rely on a scenario outline so it can run tests on each visual (example below). VisualID is the unique identifier in Power BI for that visual within the report and the ConfigPath is the relative path to the config.json file for the visual. The ConfigPath is key to understanding which visual in your Power BI report may have an error. I wish I had a more elegant solution, so I’m open to other ideas.

1) Given that we have the with the ID located in section: <Section>. Config Path:

This verifies the visual exists in the section.

2) Then the visual should have a title. Config Path:

This verifies the visual has a title as recommended for accessibility. The title does not need to be toggled on in order for this test to pass; it just needs either a literal value or conditional formatting set.

3) And ensure alt text is added to the visual if it is a non-decorative visual. Config Path:

This verifies the visual has an alt text as recommended for accessibility. If the visual is hidden on the tab order (when hidden, it’s considered decorative), this test passes by default.

4) And all visual level filters for the visual are hidden or locked in the filter pane. Config Path:

If you’d like to make sure a visual hides or locks its filters, this test will enforce that. Often, design policies will want report-level or page-level filters only. By locking or hiding visual level filters, you can improve the user experience with the filter pane. If the filter pane is hidden, this test passes by default.

Feedback

If you’ve made it this far into the article, thanks! I’ve updated the template with [instructions on GitHub](#). If you use the SampleModel to try out the visual tests, I’ve intentionally left some visuals with blank titles and blank alt text settings, so you could see the failed tests both locally and when it runs in the pipeline.

There are other tests I’d like to build, so if you’re interested in testing tab order or color contrast ratios using this technique, please contact me.

I’d love to hear your thoughts on this subject, so let me know on [LinkedIn](#), [Twitter](#), or [Mastodon](#).

Last but certainly not least, thank you Mathias for creating pbi-tools and making these type of tests possible.

Previous [Part 24: Bringing DataOps t...](#)

