Learn actionable strategies for error management modeling in Apache NiFi data pipelines, and understand the benefits of planning for error handling.

[**[](https://dzone.com/users/1055283/pieterhvmw.html)**](https://dzone.com/users/1055283/pieterhvmw.html)**by**

[**Pieter Humphrey**](https://dzone.com/users/1055283/pieterhvmw.html)

**CORE ·**

**May. 19, 21 ·**[**Big Data Zone**](https://dzone.com/big-data-analytics-tutorials-tools-news)**· Tutorial**

**Like**[**(7)**](https://dzone.com/articles/best-practices-for-data-pipeline-error-handling-in)

**Comment (1)**

**Save**

[**Tweet**](https://dzone.com/articles/best-practices-for-data-pipeline-error-handling-in)

**8.27K Views**

Join the DZone community and get the full member experience.

[**JOIN FOR FREE**](https://dzone.com/static/registration.html)

According to a [McKinsey](https://www.mckinsey.com/business-functions/mckinsey-analytics/our-insights/achieving-business-impact-with-data) report, ”the best analytics are worth nothing with bad data”. We as data engineers and developers know this simply as "garbage in, garbage out". Today, with the success of the cloud, data sources are many and varied. Data pipelines help us to consolidate data from these different sources and work on it. However, we must ensure that the data used is of good quality. As data engineers, we mold data into the right shape, size, and type with high attention to detail.

Fortunately, we have tools such as [Apache NiFi](https://nifi.apache.org/), which allow us to design and manage our data pipelines, reducing the amount of custom programming and increasing overall efficiency. Yet, when it comes to creating them, a key and often neglected aspect is minimizing potential errors.

Although many factors influence a NiFi data pipeline, three important ones are: understanding the nature of your data sources and targets, minimizing maintenance, and following best practices.

Choosing tools tailored to big data is the first step towards an efficient data pipeline. For example, let's consider a NoSQL database, such as [Apache Cassandra](https://cassandra.apache.org/), which is specifically designed to help with data availability, write-intensive workloads, and fault tolerance. As a result, many scalability, availability, replication, data locality, and throughput problems are already controlled.  When aggregating data from many sources at high volume and high velocity, there is no better landing pad for that data than Cassandra.

Once we have the good foundational choices and tools, let’s ensure we spend minimal time with boilerplate code and setting up infrastructure, as that takes us away from the code that matters. A serverless data service that needs zero setup or operations is optimal for that purpose. Then we can focus on what is most important for us: our NiFi pipeline, leaving the other technical aspects to experts.

Lastly, we must concentrate on implementing best practices. In our case, this means using proven techniques as a way to anticipate, detect and resolve potential problems in our Apache NiFi data pipeline.

This article focuses on this last point and what the author has learned working with NiFi developers. It gives you some actionable strategies that will increase the probability that your NiFi data pipeline function without unwanted interruptions and using quality data. However, these strategies are simple enough that the concepts can apply outside of the Apache NiFi, to other data pipelining or data flow tools, like Google Data Flow, AWS Kinesis, and Azure Data Factory.

**What is Apache NiFi**

Apache NiFi is an end-to-end platform that allows us to collect and act on our data pipeline in real-time. Its advantages are many. From providing a visual programming interface based on directed graphs that enables rapid development and testing, to the capacity to modify our NiFi pipeline at runtime, to its data provenance functionality that helps us track what happens with our data from beginning to end. As a consequence, businesses can start from a simple model that provides insights and results from the very beginning, and that expands into a comprehensive NiFi data pipeline.

**Why Error Handling in Your Pipelines Is Important**

Big data brings new opportunities, and also new challenges.  Apache NiFi developers cite a variety of challenges with dataflows:

1. **Systems fail:** networks, disks, software are not perfect and sometimes fail. We humans also make mistakes.
2. **Data access exceeding the capacity to consume:** data intake can be uneven, and sometimes results in capacity overload for our NiFi pipeline.
3. **Boundary conditions are mere suggestions:** data comes in many flavors, including too big, too small, too fast, too slow, corrupt, wrong, or in the wrong format.
4. **What is noise one day, becomes signal the next:** data’s value changes quickly, and we must plan for this.
5. **Systems evolve at different rates:** Apache NiFi pipelines exist to connect distributed components that were not designed to work together.
6. **Compliance and security:** an ever-evolving area, with a huge impact and where data accountability is essential for businesses to survive.
7. **Continuous improvement occurs in production:** change is necessary to succeed, and with it, the capacity to adapt.

All these cases are the sources of potential problems, and error handling can reduce their impact on business by taking preventive measures.

**An Ounce of Error Handling Prevention Beats a Pound of Debugging Cure**

**“There is an easy way and a hard way. The hard part is finding the easy way."**

**- Dr. Lloyd**

Most problems become a bit more manageable (and far less stressful) by applying some upfront analysis about what is likely to happen at runtime. In the case of a NiFi pipeline model, the first immediate boundary is given by the model itself. Thus, we can begin by separating problems according to their origin in:

1. **External:** these problems are outside of our control, as they don’t originate within the model. Examples include data not received by the model and data sent from the model but that didn’t reach its destination, for example, due to a temporary internet connection failure.  We'll about this in strategy 1 below.
2. **Internal:** these problems originate within our NiFi model and thus have the potential of being predicted and controlled.

Each NiFi sub-process consists of incoming and outgoing data and the intermediate processes.



Considering the inbound data for a moment, we can immediately identify two potential realms where issues can arise:

1. **Data intake variation:**for example, a sudden increase in the number of records received from a database that the model cannot handle properly, which can result in a data or memory overload. This problem is addressed in strategy 2 below.
2. **Data quality:** data received or generated internally that is in bad or errored condition. For example, a JSON component with a corrupted structure or with missing compulsory data. This problem is addressed in strategy 3 below.

Processes can also be the source of errors. However, not all these problems can be self-resolved. For example, wrong component designs and settings within the model can cause problems. An example is given in strategy 4.

**Error Handling Strategies for Apache NiFi**

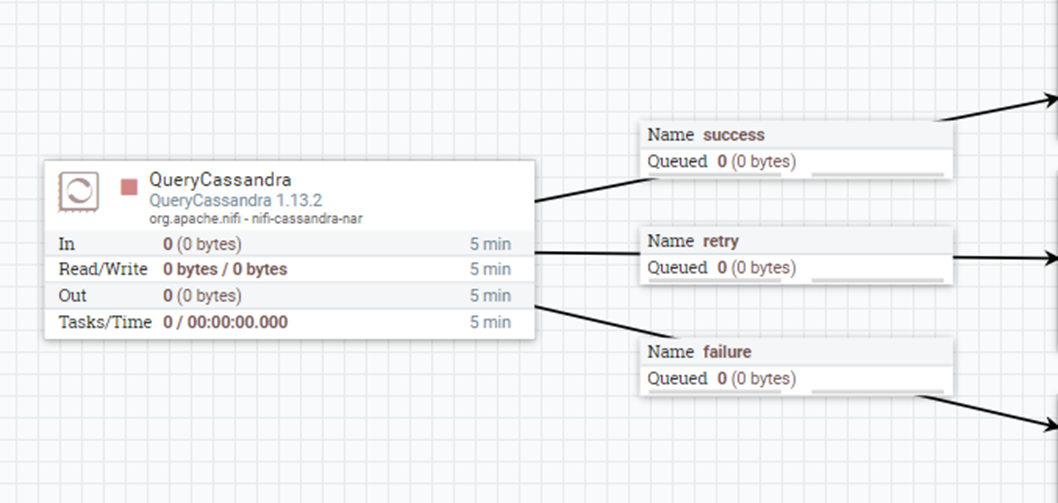
The more complex the model, the more possible sources of problems exist. Forecasting every single potential problem is, of course, impossible. Identifying the most important ones and providing self-solving solutions can greatly reduce the operational uncertainty of our NiFi pipeline and improve its [robustness](https://en.wikipedia.org/wiki/Robustness_(computer_science)).

To see how to do this analysis, we will consider four possible strategies: one external and three internal. They certainly do not cover all potential error scenarios, they are just examples that we can extrapolate from, and inform how to handle other potential failure domains.

**Error Handling Strategy 1: Retry Approach**

External sources are outside our control. Therefore, we neither know the root cause of the problem nor can we try to solve it. As a result, the best approach is to ask the source if it is back to normal or not. We will call this the retry approach.

An example would be when our model receives information from a database or data API service located on the cloud, such as [DataStax Astra DB](https://dtsx.io/3wcjyor) (which is powered by Cassandra), and suddenly we face an interrupted Internet connection. A self-solving strategy could involve using a retry, such as it is shown in the figure below:



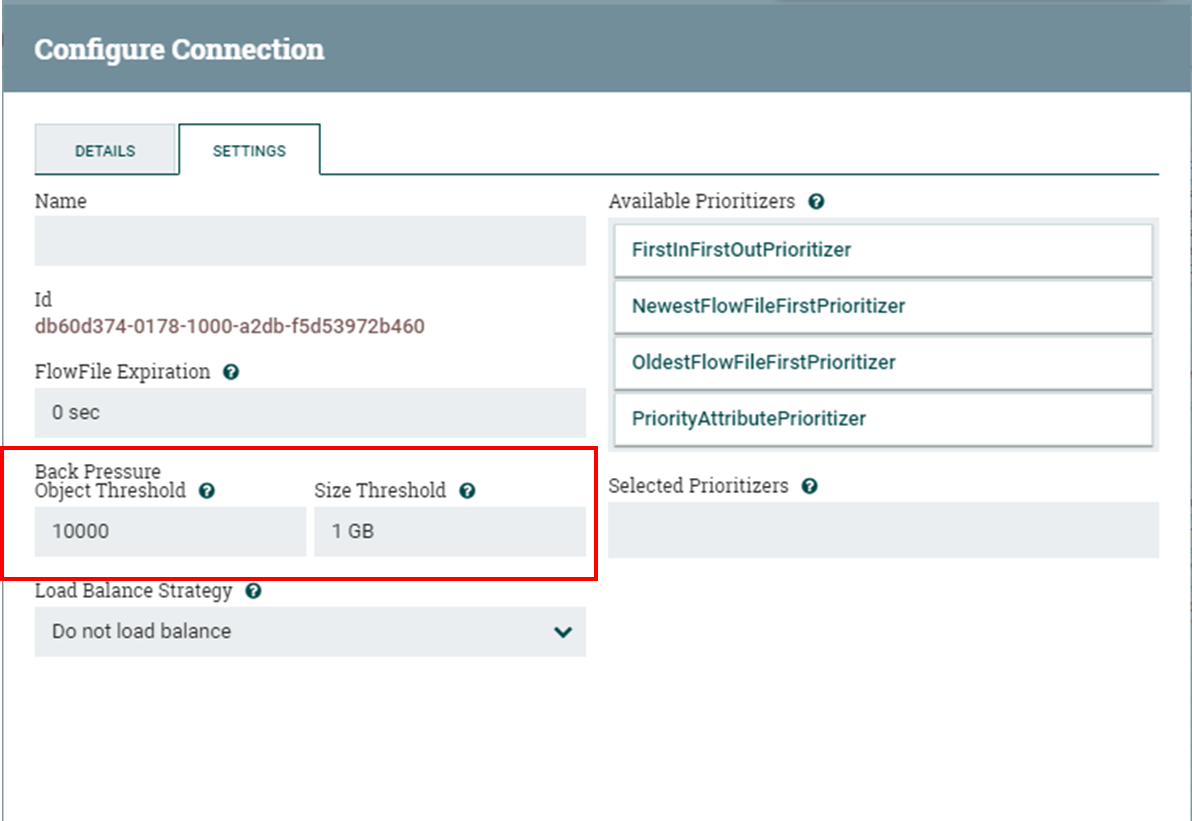
*Simple retry in Apache NiFi.*

Another way could be to create a counter, initialize it to one and then try the operation. If successful then the system continues with the process as normal. If it fails, the system increases the counter by one, checks if the counter has reached its limit, and if not, retries. If the counter reached its limit, then the system logs the error for later manual intervention.

**Error Handling Strategy 2: Using Back Pressure**

Apache NiFi provides a mechanism to manage data flow named back pressure. It consists of two thresholds, which define the maximum amount of data allowed to queue in the connector. This allows Apache NiFi to avoid data and memory overload.

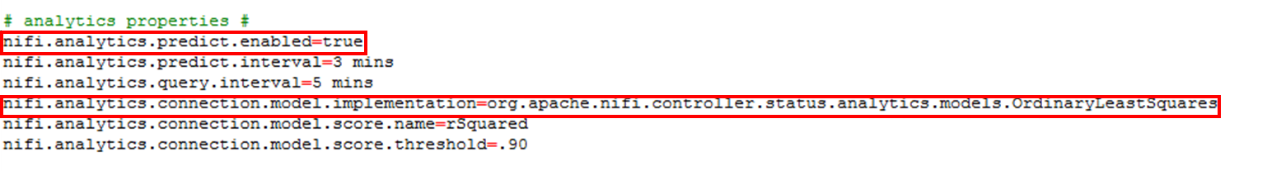
Back pressure is defined through two different values namely, “Back Pressure Object Threshold” and “Size Threshold”. The first indicates the maximum number of FlowFiles that can be in the queue before back pressure activates. The second specifies the maximum amount of data (size) that can be queued before back pressure is applied. Both values can be set from the connector’s settings section. The default values are 10000 objects and 1 GB respectively and are defined in the nifi.properties configuration file.



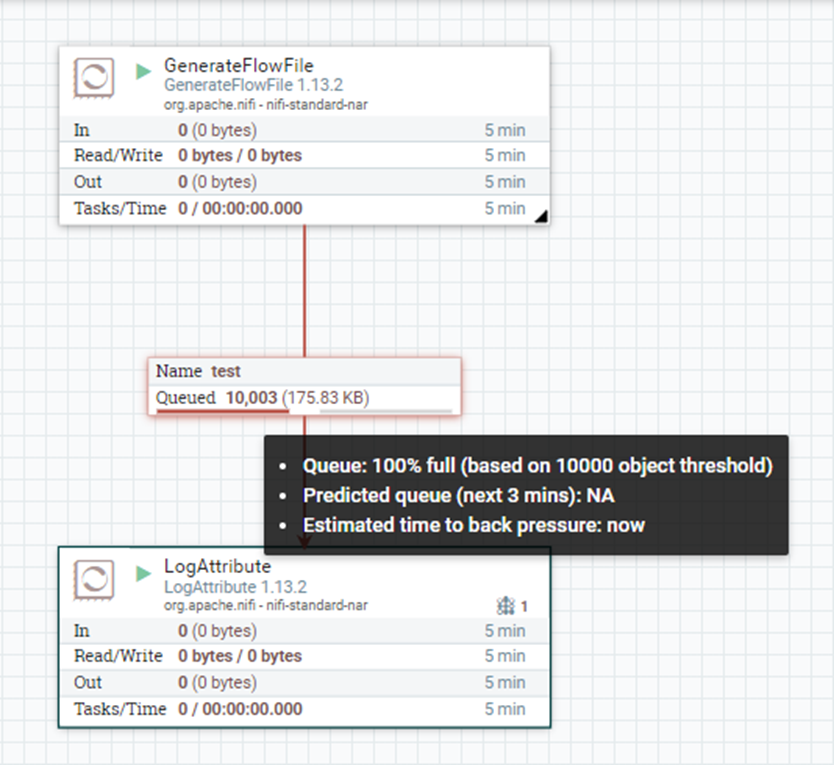
*Configuring back pressure in nifi.properties visual editor.*

However, the back pressure threshold is a flexible setting. This means that, for example, if we defined the object threshold as 10000, and the queue has 9000 objects and receives 2000; it will accept these objects reaching a value of 11000 and activate the back pressure mechanism. Once the queue releases enough messages to be below the object threshold, the connector will continue to accept more FlowFiles from the source processor.

Another tool complementing back pressure is “Back Pressure Prediction”. This function allows the system administrator to monitor the queue manually. By default, this function is not activated. To do so, we must set up to “true” the [analytics framework](https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html#analytics_framework) in the [nifi.properties](https://docs.cloudera.com/HDPDocuments/HDF3/HDF-3.5.2/nifi-system-properties/content/analytics_properties.html) configuration file.



In order to monitor the queue, we must hover over the scroll line on the connector. We will see two important values: predicted queue and estimated time to back pressure. The first value gives the estimated percentage use of the queue and the second the predicted time to back pressure activation. The figure below shows an example where random files are generated and the queue reaches the object threshold of 10000.



*Back pressure queue monitoring in Apache NiFi.*

By default, this prediction framework uses the ordinary least square method and a frequency interval of 1 minute. If we need more frequent predictions, we can set a different value for the nifi.components.status.snapshot.frequency property in the nifi.properties file.

**Error Handling Strategy 3: Using Filters**

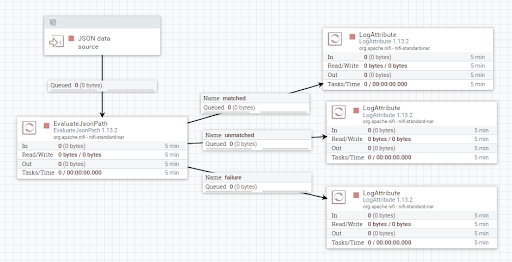
In this case, we want to check the incoming or generated data and classify it according to its quality. For example, if we connect to our Astra database via the [Stargate Document API](https://docs.datastax.com/en/astra/docs/document-api.html) and we receive the incoming data in the form of a JSON dataset, one classification could be:

* **Good:** all fields completed and in the expected format.
* **Bad:** corrupted data format. For example, incorrect fields.
* **Incomplete:** data in the correct format, but having some compulsory fields empty.

In this case, we can design a strategy that takes a different decision according to each option, such as:

* **Good:** data continues the normal process.
* **Bad:** data is not allowed to continue the normal process, and the problem is logged.
* **Incomplete:** data is not allowed to continue the normal process and it is requested again.

In Apache NiFi, this could be modeled using a set of components similar to the ones shown in the figure below. JSON data is evaluated according to certain criteria (e.g. is not null) defined using [Apache NiFi Expression Language](https://nifi.apache.org/docs/nifi-docs/html/expression-language-guide.html), and then diverted to different actions (LogAttribute in the example in the figure below) according to the result of the evaluation.

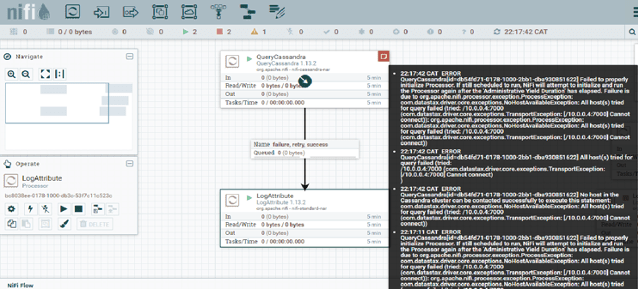


*Filtering JSON data with Apache NiFi.*

**Important note:** if the data was generated outside and received by the system, we can still consider it an internal problem as the data is now within the model’s realm.

**Error Handling Strategy 4: Manual Intervention Required!**

Some problems just need our intervention, no way around it. An example would be an outdated setting in one of the Apache NiFi pipeline’s processors. As such, the best solution is to alert your team and store the error information and data involved in a log, so you can proceed to solve the problem immediately. Issue tracking workflow automation via an issue tracker using [NiFi HTTP processors](https://ddewaele.github.io/http-communication-with-apache-nifi/) can be very helpful here, as can considering integration to Slack or PagerDuty for critical escalations.

Logging errors for manual intervention with Apache Nifi

**Conclusion**

**“Most improved things can be improved.”**

**― Mokokoma Mokhonoana**

Using data flow and data pipeline tools, in general, has many benefits: from clean modeling to faster development, to reducing custom-coded pipeline development effort. Apache NiFi offers a powerful, low code interface. Developers can prevent sticky situations by analyzing the model’s domain and anticipating potential problems, and incorporating error control logic into the pipeline design. This Apache NiFi best practice is called error handling.

Error handling planning becomes more essential as models become more complex. Apache NiFi provides a wide variety of built-in tools such as backpressure, retry options, data provenance, and much more. When used properly, a design that incorporates an upfront approach to error management translates into faster recovery, less time wasted troubleshooting and sustained performance.