Reply to reviewer's comments on
"Corral Framework: Trustworthy and Fully Functional Data Intensive Parallel Astronomical Pipelines",
by Cabral, Sánchez, Beroiz, Domínguez, Lares, Gurovich & Granito

---

We thank the reviewers for their careful reading of our manuscript and for pointing out issues, which we have attempted to address in this revised version. We have included many modifications following your suggestions. This has resulted in what we consider is a much improved manuscript. The most important additions to the text include:

i the addition of a profiler
ii a more detailed treatment of *QAI*, the proposed quality measurement metric
iii a brief example of a pipeline devised with Corral (in **Appendix A**)
iv a table that compares Corral with other pipeline generation tools (in **Appendix B**)

The main changes to the manuscript are shown in boxes, in bold typeface for newly added text, and for one correction we use strike-through text for eliminated text. The answers to questions and comments raised can be found immediately below.

Reviewer 1

> How does the development of pipelines in the context of the framework look like? What is the role of algorithm or software developers, telescope operators, researchers, etc.?

Corral presents a framework to develop pipelines in the python language. In order to provide more information on what is involved in writing a pipeline with Corral we added descriptive text to this regard as well as an appendix showing an example of a devised Corral pipeline.
To simplify the work for the software developer the main tasks concerning data management inside the database, as well as the parallel execution of steps, are set within and controlled by the framework. However, in order to use Corral the developer must first write some python code to complement with existing python modules for the purpose of establishing the: data model, data loader (to parse raw data), steps and, alerts required.
Individual roles with the pipeline will depend on the complexity of the input system, its particular characteristics (data-acquisition software., etc), and required output. Building a pipeline requires thorough knowledge of the data-stream I/O vectors and necessary code is required for each specific task according to its purpose so it is difficult to write in more general terms as to specific implementation requirements. However as noted in the text we offer to aid any potential user of Corral with advice to meet the end of using Corral to build their pipeline.

> What is actually stored in the database? Is that status information of the pipeline run, certain results, both?

The database structure is defined by the models, as explained in section **4.2** and **5.1**. The database can store raw data or metadata as well as pipeline status information. Pre-processed or final data products including scientific results can be stored but this depends on the required design. The whole data-stream could in practice be stored in the database in fact, although for most pipeline applications we do not recommend this. For example, for Toritos a pointer or the network path to each image is stored in the database and not the actual image pixel coordinates and values. Nevertheless we still consider the Toritos image data to be part of the pipeline data stream although this data is notionally stored in the database. Ultimately what is stored in the database will depend on the required design data volume & rates as well as limitations in computing infrastructure. Although a compromise may be required, we note that a wide range of possibilities are accommodated for Corral pipeline development and many of these choices should be made by the developer.

> How does the mapping from the pipeline to the hardware happen (especially when working on a compute cluster; which requires a certain level of user input)?

It is mentioned that the framework makes it easier to parallelize tasks. Please provide more details about how the framework distributes jobs over cores/nodes.

Is this only valid for embarrassingly parallel processing or is there a form of inter-process communication?

We have included a new subsection where an attempt is made to give a more thorough explanation:

4.3 Some words about Multiprocessing and distributed computing

As previously mentioned our project is built on top of a RDBMS, which by design can be accessed concurrently from a local or network computer. Every step accesses only the filtered part of the stream, so with minimum effort one can deploy the pipeline in one or more computers and run the steps in parallel. All the dangers of data corruption are avoided by the ACID (Atomicity, Consistency, Isolation and Durability) properties of the database transactions. Corral takes advantage of this technology to start a process for each Loader, Step or Alert within the pipeline, and allows interaction with the data stored in the database. If needed, groups of processes can be manually distributed on nodes of a cluster where the nodes will interact with the database remotely.
It is worth noting that any inter-processes communication is forbidden by design, regardless of whether it is read from, or written to the database. On this last particular point, the absolute isolation of the processes, is guaranteed by the MVC pattern.

Corral is able to launch more than one system process –ideally one per CPU core– each one responsible for a processor task execution (a step, loader, or alert). In order to work with several cluster nodes, a remote database needs to be configured, and afterwards each processor should be launched separately by hand (as detailed in `http://corral.readthedocs.io/en/latest/tutorial/tuto04.html#selective-steps-runs-by-name-and-groups`).
In none of the cases of parallelization there exists inter-process communication, since the MVC design pattern imposes task isolation. The pipeline shares information among processor tasks through the database exclusively.
The conclusions now include a paragraph explaining future improvements regarding distributed computing, since an implementation over these systems would be possible and yield an interesting performance boost to many pipeline applications.

> *How is the support of running components that use internal parallelism (OpenMP, MPI, etc)?*

Internally Corral's parallelism relies on multiprocessing (`https://docs.python.org/3.6/library/multiprocessing.html`). This library is included in every Python distribution and it is in charge of encapsulating objects inside an OS level process. This way any processor task which requires an external component call, for example a compiled C code which could be using MPI or OpenMP, is encapsulated in a new OS process. Now the OS scheduler is responsible of the resource management. In other words if a program with internal parallelization, of any paradigm, is being called (for example with `sh` Python library) it should run in parallel transparently.

> *Are loops implemented in the framework?*

This is clarified in the subsection:

5.3 Step Example
An example Step is displayed below, where a simple data transformation is conducted. Where the step takes a set of sources and transforms their (x,y) coordinates to (RA,Dec): lines 10-12 show definitions for Class-level attributes, which are responsible for this query. The framework retrieves the data and serves it to the `process` method (line 14), which executes the relevant transformation and loads the data for each source.
(...)

The framework implements loops internally in every step and alert. As shown in the examples provided each of these processors contains several class methods, which select the necessary material for execution. In particular the framework will expect from these methods the delivery of generators, which are Python's iterable objects specifically designed for loops. Once Corral gets these generators it will start looping over them, without user interaction.

> Does the framework handle failing processes and retries? Can a minimum (e.g. 5% of jobs may fail without marking the whole pipeline as such) be specified per task?

Retry attempts are not in Corral's design. But this doesn't mean that tasks cannot fail, or that the pipeline is not failure tolerant. There are no utilities specifically built for error tolerance, but the framework is flexible enough to implement error states. Specific retry Steps can be incorporated such as shown by the lines 526–563 in: https://github.com/toros-astro/toritos/blob/master/toritos/steps.py#L526-L563, where `StepRetryAstrometry` implements retries inside the Toritos pipeline.
In any case, testing and quality specific tools included in the framework should aid the developer in the error detection. There is a new subsection with more detailed information related to this question:

4.4.5 Final words about Corral quality
The framework does not contain any error backtrace concept, or retry attempts in processing. Each processor should be able to handle correctly the required information on its conditions. It is implicitly expected that the quality tools offered serve to unveil code errors as well.
If the pipeline's developer achieves a high code coverage and is able to test enough data diversification, the possible software bugs can decrease substantially, up to 80% (Jeffries & Melnik 2007).

There is though a second answer to this question, that also brings highly technical discussion related to Object Oriented programming and the nature of the MVC pattern. By design any processor task interacting with the database is linked through a session connection (derived from SQLAlchemy ORM API) which is in charge of querying and committing the changes to the database. Since RDBMS follow ACID, every task can finish with two possible outcomes: success or complete failure. Such a restrictive behavior (necessary in order to preserve data integrity) is transferred to the processor tasks.
Corral features in its base classes for processor tasks several methods, some of them are specific to Python's class *context managers*, like the one called `teardown`. This method is executed always, and receives as parameter the error state of the class processing outcome. At the developer's discretion some special behavior to this `teardown` can be implemented, depending on the error associated to the class processing outcome. From this, it is possible to derive an error tolerance behavior powerful enough to cover most pipeline error cases.

> What type of community is Corral mostly useful for? Two extrema are large, international projects with hundreds of users and shared infrastructure on one hand, and on the other end is the single astronmers wanting to run a small pipeline on their laptop.

Depending on the underlying database (and of course of hardware resources available) Corral is suitable for a wide spectrum of pipeline sizes.
A good choice of the database implementation useful for a single developer who needs to create small local experiments in short periods of time could be SQLite. SQLite has proven valuable to the authors, in daily pipeline design and prototyping due to its simple configuration and lightweight resource consumption.
At the same time the framework is also suitable for big data reduction pipelines which could be built on top of multicore machines, for example by using a much more powerful SQL vendor. In this case PostgreSQL implementation has proven to be a great tool for heavy duty processing in the authors' experience and can serve a great number of requests without noticeable lagging.
To clarify this, a paragraph has been added inside section **4.1 The Underlying Tools: Python and Relational Databases**.

Another key requirement is the storage and retrieval of data in multiple processes, which led us to use *Relational Databases Management Systems* (RDBMS). Relational Databases are a proven standard for data management and have been around for more than thirty years. They support an important number of implementations and it is worth mentioning that amongst the most widely used are Open Source, e.g., PostgreSQL.
SQL is a powerful programming language for RDBMS and offers advantages in data consistency, and for search queries. ***SQL* has a broad spectrum of implementations: from smaller, local applications, accessible from a single process, like SQLite (Owens & Allen 2010), to distributed solutions on computer clusters, capable of serving billions of requests, like Hive (Thusoo et al. 2009). This plethora of options allows flexibility in the creation of pipelines, from personal ones, to pipelines deployed across computing clusters hosting huge volumes of data and multiple users. Inside the Python ecosystem, the SQLAlchemy library offers the possibility of designing model Schema in a rather simple way while at the same time offering enough flexibility so as to not cause dependence issues related to specific SQL dialects. Thus offering a good compromise to satisfy different needs of the developer.**

> What properties should a processing pipeline have to be portable to Corral? And maybe more important for interested developers: what work flows do not map the design well? Of course, this list is not limitative. In the introduction, you give a very short overview of a several pipeline frameworks and pipelines (Kira, Apache Commons Pipeline, and several custom telescope pipelines). To give an idea how Corral compares to and differs from already existing systems, please add a section that discusses for several existing frameworks how they differ in design, functionality and/or background idea from Corral. For this, it is important to look at more than only astronomy (bioinformatics for example seems to have an active usage of pipeline frameworks) is. It may be a good idea to quickly skim through https://github.com/pditommaso/awesome-pipeline. Naming a few examples that at quick glance may be worth comparing to (either because they are widely used or because the description seems to be somewhat close to what you do): Luigi, Airflow, Arvados, Kepler, or OPUS (`http://www.stsci.edu/institute/software_hardware/opus/`*), but please feel free to make your own selection.*

Every processing pipeline capable of being separated in a loader, several steps and optionally alerts is portable to Corral, since MVC is based on concept separation this is the only true restriction.

Corral is less powerful when the pipeline is about processing a single batch of data just once. The effort of splitting the processing stages brings no profit unless the pipeline is being fed regularly with new data. Corral is extremely useful when data processing is not needed in real time, this should not be confused with the previous statement.

A comparative table is now on **Appendix B**, showing how Corral is different from the suggested frameworks. Based on this table the subsection **7** was developed, highlighting the features that Corral takes advantage of, without hiding its weaknesses related to other software.

1 COMPARISON WITH OTHER PIPELINE FRAMEWORKS

The two main differences between Corral and other similar projects are now explained. (A comparison of other alternatives is shown in Appendix B): First, to our knowledge Corral is the first implementations of a pipeline framework that uses MVC; and second the quality integration metrics that give an indication of the trustworthiness of the resulting pipeline.

The use of the MVC design standard imposes the following processing tasks (Loader, Steps and Alerts) result in a strong isolation condition: every processing stage only interacts with filtered data according to specific criteria with no bearing on information of the source or data destination. A major advantage of isolation is the natural parallelization of processing tasks since no "race condition" is created. Regarding real-time processing this pattern can be inconvenient, since batches of data are processed asynchronously, leading to random ordering of data processing and writing onto the Database.

Corral features "integration quality" utilities as an important tool set that builds confidence on the pipeline. This works when unit-tests are available for the current pipeline and in these cases Corral can automatically generate reports and metrics to quantify reliability. Corral was designed to optimize pipeline confidence in terms of some global notion of quality, which implies revision of data in each processing stage. Pelican is the only project integrating tools for pipeline testing, but does not include extra functionalities based on this concept. In many other aspects as depicted in appendix B, Corral is similar to other alternatives. A majority of these alternative also use Python as the programing language mainly in order to make use of its vast libraries for data access and manipulation, across multiple formats.

> Apart from those two main points, I have several questions / comments concerning the manuscript: Stimulating astronomers (or scientists in general, for that matter) to write readable, testable and maintainable code is commendable effort and I am happy to see that being part of this project. Software quality can be expressed in many different ways. The formula on page 4 seems however somewhat arbitrary. Please add a short comment on why you focus on clean code and testing and not on things like performance, errors and exceptions for example. One comment that should be made is that this method measures how completely the code is tested. It does not say anything on the quality of the tests itself. It is very easy to come up with tests that make the QAI score high by covering large amounts of code but do not reliably test functionality. Only relying on this number could give a false sense of confidence.

Why do you scale the syntactic part of the analysis with an exponential function? You implicitly seem to assume that T/PN ¡= 1 (since a larger number could make the QAI become large than 1). I don't see a reason why a processor could have only one unit test associated to it.

Thanks for this remark, it was really helpful since it provided a new insight on the proposed metric. Following the suggestions the formula has been revised.

The previous version had the following looks:

$$QAI = \frac{1}{2} \times \frac{TP \times Cov \times \frac{NT}{NP}}{\left(1 + exp\left(\frac{N_{SError}}{\tau \times N_f}\right)\right)}$$

The *QAI* was splitted in two pieces, separating style from the other metrics by explicitely writing the attenuation factor γ in the denominator.

$$QAI = \frac{\Theta \times \Lambda_{Cov} \times R_{PT}}{\gamma}$$

The test passing part of the equation TP was transformed in a Heaviside function Θ which again is 1 if every test passes, and 0 if anyone fails. We also encoded code coverage in Λ_{Cov}, and modified the ratio of NT/NP, with the introduction of R_{PT}: the ratio of processors being tested. Basically we modified this quantity so it asks at least one test per processor in order to score 1. In case of more than one test per processor it will remain at the maximum score, removing the "greater than one" problem.

The style errors moved to a separated function γ, which is identical in structure as the denominator part of previous

formula,

$$\gamma = \frac{1}{2} \times \left(1 + exp\left(\frac{N_{SError}}{\tau \times N_f}\right)\right),$$

in this case we also decided to put the tolerance factor.
Generally speaking, this metric *QAI* intends to account for the meeting of requirements that the pipeline is achieving. Since quality is subjective, and several interpretations are possible, we focused in testing the software that the developer is writing. This means that a QAI close to one is the same as to say "the pipeline is doing exactly what the developer was trying to do", this should be true even in corner cases where the pipeline is raising exceptions, since that may be the developers prerogative as well.
So we also added to the manuscript the following paragraph:

> This index aims to encode in a single figure of merit how well the pipeline meets the requirements specified by the user. We note that this index represents a confidence metric. Thus a pipeline could be completely functional even if the tried tests all fail, or if no tests are yet written for it. The *QAI* index attempts to answer the question of pipeline reliability and whether a particular pipeline can be trustworthy. It should not be confused with the pipeline's speed, capability, or any other performance metric.

To test the quality of tests itself is a difficult task, and as any other quality measure is subjective. Our approach is based on Code Coverage, as it focuses on the information that tests provide, and how complete it is. The authors agree that it is possible to fool the metric by writing nonsense tests, and the confidence levels could be also biased depending on the case. However there are no flawless quality determinations, at least not simple ones. This is the main reason the coverage and style were adopted, they are simple and easy to quantify (specially for the framework) and if testing is done faithfully they provide a good idea of the risks of deploying the pipeline for production.
In order to clarify by example the following case could be considered. The idea is to compare the reliability on two software projects: first consider the case where build is passing, but coverage is 15%; secondly, imagine a project which coverage is 95% but build is not passing. If we think on which project is safer to deploy, the authors agree that the second would be necessarily preferred, since at least we have information regarding the 95% of the project, and is quite likely that the build error is a minor one.
Regarding performance measurements as quality metrics we included the following paragraph in the subsection **4.4.5 Final words about Corral quality**:

> The framework does not contain any error backtrace concept, or retry attempts in processing. Each processor should be able to handle correctly the required information on its conditions. It is implicitly expected that the quality tools offered serve to unveil code errors as well.
>
> If the pipeline's developer achieves a high code coverage and is able to test enough data diversification, the possible software bugs can decrease substantially, up to 80% (Jeffries & Melnik 2007).

Regarding style we use the attenuation factor γ since an style error is not as critical as other kind of faults, and a the scaling parameter τ gives freedom to the user to decide if its necessary to follow a strict standard (authors follow PEP08 only) or to take less seriously the style mistakes.
This freedom is useful in different environments, for example a simple pipeline that runs in a single laptop where the developer is the customer may be less important the style guide. In large projects running inside institutions, for example in a cluster, where a lot of scientists are reading the code to work on a daily basis, the style standard can be a critical tool to share, maintain, develop and understand running code inside a community.

> In the presented code example, it is not really clear to me how the framework decides what the order of steps is. This is of course due to the fact that there is only 1 step.

Steps are processors which work by starting a transaction in the database. More clearly, the first thing a Step does is to query the database asking for processing material, and if no data was found nothing happens. A different way of seeing this is “Steps operate whenever data in the right conditions is present”. Following this the order of Steps is irrelevant, since they happen if given the needed conditions, but there exists no order between them.
There exists the possibility of grouping Steps, in order to state explicitly that a subset of Steps can run asynchronously, and the user could run this groups in any order he desires by manually running

```
python in_corral.py run --step-groups StepGroupName
```

> What is the overhead of the framework in terms of resource utilization and how do you expect that to behave when scaling up to many processes?

A paragraph regarding this issue can be found in the new section **6 Corral: Under the hood**.

(...) As mentioned the most basic functionality of Corral is to find files based in only one environment configuration. This brings to the developer clear rules to split the functionality in well defined and focused modules.
According to measuerd estimates, typically 1.5–2 s of overhead is required for any executed command, depending on the number of processes spawned. Almost 100% of the overhead is spent before the true command logic is executed. An interesting point to this is that the running mode of commands strongly affects the execution time (’out’ mode is much faster than ’test’ with in-memory database; and test is much faster than ’in’ mode). A final caveat to performance worth mentioning are some external factors for potential bottlenecks, like database location, I/O, Operating System configuration, or any hardware problems.

We have registered small overhead caused by the framework. Based on simple profiling we conclude that the framework takes at most 10 import calls of overhead prior to start processing data. The true overhead though, is because database configuration, queries, memory loading time, the number of processes running at the same time, and of course the underlying hardware.

> Finally, I have a few textual comments:

- *”In our approach we suggest a simple unit testing to check for the status of the stream before and after every Step, Loader or Alert”. I am not sure if I get what you mean here. What does ”unit testing” mean in the context of the status a data stream?*
- *Please check that the authors of citations are sensible (not Initiative et al., for example). You may actually want to put a web link in a footnote in some of those cases.*
- *page 4: ”Where TP is 1 if test passes or 0 if it fails” -¿ ”Where TP is 1 if all tests pass or 0 if at least one test fails.”*
- *I am not sure what you mean by the first part (until the comma) in the sentence ”Given the wide range of available libraries, Corral includes the ability to automatically generate documentation, creating a manual and quality reports in Markdown syntax (...)”.*

Firstly we define a Step, Loader or Alert as a processing unit. Given this, from a stream point of view the unit nature of the tests means that is necessary to create and destroy any information contained in the stream before and after every test.
In order to further explain this some comments are now present in section **4.4 Quality – Trustworthy Pipelines**

(. . .) In our approach we suggest a simple unit testing approach to check the status of the stream before and after every *Step*, *Loader* or *Alert*.
Because tests are unitary, Corral guarantees the isolation of each test by creating and destroying the stream database before and after execution of the test. If you feed the stream with a sufficient amount of heterogeneous data you can check most of the pipeline's functionality before the production stage. Finally we provide capabilities to create reports with all the structural and quality assurance information about the pipeline in convenient way, and a small tool to profile the CPU performance of the project.

Regarding citations we thank you for the suggestions, they were addressed, and now the authors appear when possible, and if not a link is available.
The correction regarding the test passing quantity has been addressed.
The phrase related to existing libraries is now modified and the text reads like the following:

~~Given the wide range of available libraries,~~ Corral includes the ability to automatically generate documentation, creating a manual and quality reports in Markdown (. . .)

Reviewer 2
Major Revisions:

> 1a. Performance metrics on case study and/or on other deployments.
Besides code correctness and quality, a streaming data reduction/processing framework should be judged on its performance. As a potential user I would like to know how it scales across cores, what is the overhead of the SQL database? When will the database become the bottleneck (if ever)? How does Corral compare to other frameworks?

This is an interesting remark since it spans some of the changes in the revised version.
The database overhead is quite variable, and a thorough quantification is technical and out of the scope of our work. However the framework makes no difference among RDBMS vendors, and the same transaction on several different database implementations or configurations (from DB or OS, or underlying hardware) will produce variable profile results. Nevertheless a small discussion on this issue is now included in section **4.1 The Underlying Tools: Python and Relational Databases**.

(. . .) *SQL* is a powerful programming language for RDBMS and offers advantages in data consistency, and for search queries. ***SQL* has a broad spectrum of implementations: from smaller, local applications, accessible from a single process, like SQLite (Owens & Allen 2010), to distributed solutions on computer clusters, capable of serving billions of requests, like Hive (Thusoo et al. 2009). This plethora of options allows flexibility in the creation of pipelines, from personal ones, to pipelines deployed across computing clusters hosting huge volumes of data and multiple users. Inside the Python ecosystem, the SQLAlchemy library offers the possibility of designing model Schema in a rather simple way while at the same time offering enough flexibility so as to not cause dependence issues related to specific SQL dialects. Thus offering a good compromise to satisfy different needs of the developer.**

Once the database vendor is chosen it is possible to profile it by using its specific tools (each vendor provides tools, and they are not standardized). However, if a specific query is causing a steep performance drop there are ways to

overcome it. SQLAlchemy provides the option of avoiding the whole library interface, and to execute SQL syntax code directly into the database, seizing its whole relational engine power. This is also noted in section **4.1** and **4.3 Some words about Multiprocessing and distributed computing** where is stated:

> As previously mentioned our project is built on top of a RDBMS, which by design can be accessed concurrently from a local or network computer. Every step accesses only the filtered part of the stream, so with minimum effort one can deploy the pipeline in one or more computers and run the steps in parallel. All the dangers of data corruption are avoided by the ACID (Atomicity, Consistency, Isolation and Durability) properties of the database transactions. Corral takes advantage of this technology to start a process for each Loader, Step or Alert within the pipeline, and allows interaction with the data stored in the database. If needed, groups of processes can be manually distributed on nodes of a cluster where the nodes will interact with the database remotely.
> It is worth noting that any inter-processes communication is forbidden by design, regardless of whether it is read from, or written to the database. (...)

The database will introduce a bottleneck if working with a cluster of computers the remote connection or the disk throughput is not suitable for handling the traffic generated by transactions.
Regarding the asked comparison we have added a table reporting several differences and similarities among frameworks including Corral in **Appendix B**.

> 1b. Performance/Hardware utilization monitoring.
It would be nice to mention if and how one can obtain monitoring metrics of the pipeline itself to determine the user's bottlenecks. Are there bottlenecks in a certain loader or step? How well am I utilizing my underlying hardware, how does it map to the underlying hardware?

Thanks for the useful suggestion. Now we have added to the manuscript a new subsection **4.4.4 Profiling**:

> Corral offers a deterministic tool for the performance analysis of CPU usage at a function level during the execution of unit tests. It is worth noting that in case a pipeline shows signs of lagging or slowing down, running a profiler over a unit test session can help locate bottlenecks. However for rigorous profiling, real data on real application runs should be used, instead of unit testing.
> Another kind of profiling at the application level could be carried out manually using existing Python ecosystem tools such as *memory_profiler* (memory line level deterministic profiling), `statsprof.py` (statistic function level profiling), o *line_profiler* (line level deterministic) amongst other tools. We note that although some application level profiling tools are included or suggested, Corral was never intended to offer a system profiling tool. Nor does it claim to offer data base profiling, or I/O, energy profiling, network profiling, etc.

> 2. Quality metric (QAI) justification
The quality assurance index is interesting both for the framework and a users implementation. However, I feel more insight into the relation presented in the QAI equation would be instructive, especially the distribution of weights to the various contributors like the style errors.

Thanks for this comment, it was also remarked by the first reviewer and a revised version of *QAI* has been addressed both in the text and first reviewer's answers. Here we reproduce the explanation regarding this proposed quality metric.
The previous version had the following looks:

$$QAI = \frac{1}{2} \times \frac{TP \times Cov \times \frac{NT}{NP}}{\left(1 + exp\left(\frac{N_{SError}}{\tau \times N_f}\right)\right)}$$

The *QAI* was splitted in two pieces, separating style from the other metrics by explicitly writing the attenuation factor γ in the denominator.

$$QAI = \frac{\Theta \times \Lambda_{Cov} \times R_{PT}}{\gamma}$$

The test passing part of the equation TP was transformed in a Heaviside function Θ which again is 1 if every test passes, and 0 if anyone fails. We also encoded code coverage in Λ_{Cov}, and modified the ratio of NT/NP, with the introduction of R_{PT}: the ratio of processors being tested. Basically we modified this quantity so it asks at least one test per processor in order to score 1. In case of more than one test per processor it will remain at the maximum score, removing the "greater than one" problem.
The style errors moved to a separated function γ, which is identical in structure as the denominator part of previous formula,

$$\gamma = \frac{1}{2} \times \left(1 + exp\left(\frac{N_{SError}}{\tau \times N_f}\right)\right),$$

in this case we also decided to put the tolerance factor.
Generally speaking, this metric *QAI* intends to account for the meeting of requirements that the pipeline is achieving. Since quality is subjective, and several interpretations are possible, we focused in testing the software that the developer is writing. This means that a QAI close to one is the same as to say "the pipeline is doing exactly what the developer was trying to do", this should be true even in corner cases where the pipeline is raising exceptions, since that may be the developers prerogative as well.
So we also added to the manuscript the following paragraph:

> This index aims to encode in a single figure of merit how well the pipeline meets the requirements specified by the user. We note that this index represents a confidence metric. Thus a pipeline could be completely functional even if the tried tests all fail, or if no tests are yet written for it. The *QAI* index attempts to answer the question of pipeline reliability and whether a particular pipeline can be trustworthy. It should not be confused with the pipeline's speed, capability, or any other performance metric.

To test the quality of tests itself is a difficult task, and as any other quality measure is subjective. Our approach is based on Code Coverage, as it focuses on the information that tests provide, and how complete is it. The authors agree that is possible to fool the metric by writing nonsense tests, and the confidence levels could be also biased depending on the case. However there are no flawless quality determinations, at least not simple ones. This is the main reason the coverage and style were adopted, they are simple and easy to quantify (specially for the framework) and if testing is done faithfully they provide a good idea of the risks of deploying the pipeline for production.
In order to clarify by example the following case could be considered. The idea is to compare the reliability on two software projects: first consider the case where build is passing, but coverage is 15%; secondly, imagine a project which coverage is 95% but build is not passing. If we think on which project is safer to deploy, the authors agree that the second would be necessarily preferred, since at least we have information regarding the 95% of the project, and is quite likely that the build error is a minor one.
Regarding performance measurements as quality metrics we included the following paragraph in the subsection **4.4.5 Final words about Corral quality**:

> (...) It is implicitly expected that the quality tools offered serve to unveil code errors as well.
> If the pipeline's developer achieves a high code coverage and is able to test enough data diversification, the possible software bugs can decrease substantially, up to 80% (Jeffries & Melnik 2007).

Regarding style we use the attenuation factor γ since an style error is not as critical as other kind of faults, and a the scaling parameter τ gives freedom to the user to decide if its necessary to follow a strict standard (authors follow PEP08 only) or to take less seriously the style mistakes.

This freedom is useful in different environments, for example a simple pipeline that runs in a single laptop where the developer is the customer may be less important the style guide. In large projects running inside institutions, for example in a cluster, where a lot of scientists are reading the code to work on a daily basis, the style standard can be a critical tool to share, maintain, develop and understand running code inside a community.

Minor Revisions:

> 1. Other work.
Only one pipeline framework mentioned outside of astronomy and two pipeline frameworks in total, there are many more. (ZeroMQ, Storm, Pelican, ...) Where does Corral fit into this spectrum?

A comparative table is included in this newer version as **Appendix B**. Regarding some of the software projects mentioned we found that:

- Pelican was quite similar with some of the other frameworks in the table, and we decided not to included in the comparison for this reason, since it was not adding much more information.
- Storm is a low level pipeline framework for distributed computing. We are considering a future re-implementation of Corral's scheduler over Apache Storm, as explained in the Conclusions.
- ZeroMQ is an implementation of AMPQ (Advanced Message Queuing Protocol) –similar *rabbitMQ*– and it is useful for any kind of data or process communications, not just pipelines. Although it may be of some use for pipeline creation, we think it may lack the necessary structure a framework would require under the standard definition.

We can summarize that **Corral is a framework that follows the MVC pattern, by mounting a data stream on top of a RDBMS. This is the main difference between Corral and most other known pipeline framework software.**

> 2. Grammar and syntax
The abstract alone contains grammatical errors and sentences that should be corrected/restructured:
- The programmer is referred to as "him"
- ...by avoiding the programmer deal with... (The whole sentence should be restructured)
- This kind of programs are chains of ... -¿ Processing pipelines are chains of ...
- Besides data transformation a pipeline can also reduce data (potentially)

We have attempted to address the suggestions, and the text has been corrected or modified.
Changes in the abstract are shown below:

Data processing pipelines represent an important slice of the astronomical software library that include chains of processes that transform raw data into valuable information via data reduction and analysis. In this work we present Corral, a Python framework for astronomical pipeline generation. Corral features a Model-View-Controller design pattern on top of an SQL Relational Database capable of handling: custom data models; processing stages; and communication alerts, and also provides automatic quality and structural metrics based on unit testing. The Model-View-Controller provides concept separation between the user logic and the data models, delivering at the same time multi-processing and distributed computing capabilities. Corral represents an improvement over commonly found data processing pipelines in Astronomy since the design pattern eases programmers from dealing with processing flow and parallelization issues, allowing them to focus on the specific algorithms needed for the successive data transformations and at the same time provides a broad measure of quality over the created pipeline. Corral and working examples of pipelines that use it are available to the community at https://github.com/toros-astro.

> *I recommend a read through of the paper on grammar and syntax by a native English speaker.*

We have attempted to make the grammar and syntax clearer.

References

Jeffries R., Melnik G., 2007, IEEE Software, 24, 24
Owens M., Allen G., 2010, SQLite. Springer
Thusoo A., et al., 2009, Proceedings of the VLDB Endowment, 2, 1626