Measuring the Power of Learning.™

Research Report
ETS RR–18-14

A Robust Microservice Architecture for Scaling Automated Scoring Applications

Nitin Madnani
Aoife Cahill
Daniel Blanchard
Slava Andreyev
Diane Napolitano
Binod Gyawali
Michael Heilman
Chong Min Lee
Chee Wee Leong
Matthew Mulholland
Brian Riordan

December 2018

ETS Research Report Series ISSN 2330-8516

RESEARCH REPORT

A Robust Microservice Architecture for Scaling Automated Scoring Applications

Nitin Madnani,[1] Aoife Cahill,[1] Daniel Blanchard,[2] Slava Andreyev,[1] Diane Napolitano,[1] Binod Gyawali,[1] Michael Heilman,[3] Chong Min Lee,[1] Chee Wee Leong,[1] Matthew Mulholland,[1] & Brian Riordan[1]

1 Educational Testing Service, Princeton, NJ
2 Parse.ly Analytics, New York, NY
3 Civis Analytics, Chicago, IL

We present a microservice architecture for large-scale automated scoring applications. Our architecture builds on the open-source Apache Storm framework and facilitates the development of robust, scalable automated scoring applications that can easily be extended and customized. We demonstrate our architecture with an application for automated content scoring.

Keywords Automated scoring; machine learning; Apache Storm; microservices

doi:10.1002/ets2.12202

Automated scoring of written or spoken student responses has received a lot of attention from the natural language processing (NLP) research community in the last few years (Burrows, Gurevych, & Stein, 2015; Dong & Zhang, 2016; Dzikovska et al., 2013; Farra, Somasundaran, & Burstein, 2015; Liu et al., 2014; Loukina, Zechner, Chen, & Heilman, 2015; Shermis & Burstein, 2013). However, building large-scale automated scoring applications for a production environment poses a different set of challenges. Reliability, scalability, and flexibility are all desirable characteristics of such an application, and it is often easy to optimize one of these at the expense of others. Automated scoring pipelines have many moving parts, have multiple stakeholders, and typically involve numerous teams within an organization. Therefore it is critical that they be developed within a framework that can support the challenges of competing demands.

The reliability (stability) of a production-based automated scoring application is crucial. An unstable system (e.g., if it crashes, fails to deliver scores, or gives feedback with low latency) is effectively useless to most test takers and clients. At the same time, the ever-increasing number of student responses that are passed to large-scale scoring applications puts pressure on the applications to scale quickly to increased volumes of data (particularly when the increase is spiked, as in during a test administration). Like with other real-time services, clients expect (and often have contractual guarantees) to receive scores (or automated feedback) from a scoring application in a given amount of time and will not tolerate a slowdown in performance because of high load. Business demands also require that new functionality or customizations be frequently added to keep up with new populations and forms of writing. This involves multiple software developers who all need to be able to contribute to the code base concurrently. A modular design facilitates efficient development by multiple developers (Burstein & Marcu, 2000). A further complication within an organization can be that multiple programming languages are used (particularly where a research group is responsible for continued enhancement of the features of such an application).

In this report, we present an architecture for automated scoring applications, designed to be scalable, robust, and flexible. In the Apache Storm section, we build on an open-source framework and extend it to address some limitations. Then, in the A Storm-Based Architecture for Automated Scoring section, we show how the new architecture can be easily instantiated for multiple automated scoring applications and compare it to a more traditional architecture in terms of speed. In the Content-Scoring Application section, we exemplify the microservice architecture using an application that scores content knowledge along with a Web interface where users provide answers to short-answer questions and receive scores in real time.

Corresponding author: N. Madnani, E-mail: nmadnani@ets.org

Apache Storm

The architecture we describe leverages the open-source, distributed, message-based computation system called Apache Storm.[1] Although frameworks like MapReduce and Hadoop make it significantly easier to conduct batch processing of large amounts of data, they are not designed to support real-time processing, which has a fundamentally different set of requirements.

We briefly introduce some of the Apache Storm terminology that is necessary to follow this report. Storm is a *stream-processing framework* (Stonebraker, Çetintemel, & Zdonik, 2005); that is, it performs computations over data as the data enter the system, for example, computing features and scores for written or spoken student responses in real time. Compared to a batch-processing paradigm, where computational operations apply to an entire data set, stream processors define operations that are applied to each individual datum as it passes through the system. A difference between Storm and Spark, another stream-processing framework, is that Spark performs data-parallel computations, whereas Storm performs task-parallel computations. A Storm application is comprises three major components:

1. *Spouts* produce the data streams to be processed by the application.
2. *Bolts* consume data streams, apply operations to them, and produce the results of the operation as additional data streams.
3. A *topology* is a network of spouts and bolts, usually organized as a directed acyclic graph with edges denoting data dependencies between the bolts. A final bolt is added at the end of the graph to produce the final result.

Next, we discuss the significant advantages that Storm provides for building applications that not only process data streams at scale but also provide results in real time — the use case that we describe in this report (see the Content-Scoring Application section).

Scalability

Storm can scale to millions of data streams per second simply by increasing the parallelism settings in the topology and adding more compute nodes to the cluster. It also automatically handles running bolts in parallel to maximize throughput.

Robustness

Storm also provides strong guarantees that every data stream *will* be processed and never dropped. In case of any faults during a computation, for example, a hardware failure, Storm automatically reassigns tasks to other nodes to ensure that an application can run forever (or until explicitly stopped).

Programming Language Agnosticism

Storm topologies, bolts, and spouts can be defined in *any* programming language. This is particularly important in NLP, where high-quality open-source libraries are available in many programming languages, for example, StanfordNLP/Java (Klein & Manning, 2003), NLTK/Python (Bird, Klein, & Loper, 2009), and WordNet::Similarity/Perl (Pedersen, Patwardhan, & Michelizzi, 2004).

Ease of Customization

Because bolts can be shared across topologies, a central repository can easily contain all bolts. Each new instantiation of a Storm topology (e.g., an automated scoring application) can be easily created by selecting relevant bolts and defining the information flow through them. This also allows for convenient "plug and play" experiments with different NLP core components, such as parsers or taggers.

A Storm-Based Architecture for Automated Scoring

In their simplest form, automated scoring applications can be seen as machine learning applications where an automated scoring model is (a) trained on a large set of human-scored responses; (b) evaluated for accuracy, fairness, and validity on held-out data; and (c) deployed in a production setting requiring high throughput and low latency. Although this is a fairly simplified view of such applications—for example, it ignores monitoring components that automatically detect anomalous conditions, such as drifts in machine scores and application failures—it should suffice for the goal of this report.

Although Storm affords several advantages to NLP applications, it has some gaps in the case of building automated scoring applications:

- Although Storm has support for multiple programming languages, it does not actually provide any reliable Perl and Python bolt and spout implementations out of the box.
- Storm does not provide a way to identify and group data streams; that is, there is no way to say "I want the output from all the bolts in the topology for the same student response together."
- If a response causes an exception in one of the bolts (e.g., the parser times out because a response is too long), Storm will retry the same response forever, instead of bypassing its processing in any downstream bolts.

Our architecture extends Storm to address all of the preceding issues. The core of our enhanced architecture is a custom implementation of a Storm bolt that we term a `FeatureService`. `FeatureService` is an abstract class with implementations in Java, Perl, and Python. Each object inherited from this class represents a *microservice* that performs a simple, discrete task on the data streams that pass through it. Our Perl and Python implementations of `FeatureService` use the `IO::Storm`[2] and the `streamparse`[3] open-source libraries, respectively, and bring them up to par with the corresponding Storm-native Java implementation.

`FeatureService` encapsulates an extremely easy-to-use interface for developers: After subclassing `FeatureService` in their preferred programming language, they only need to implement two methods: `setup()` to load any resources required and `calculate()` to perform the processing. Finally, `FeatureService` implements data stream grouping: Developers specify which other feature services are its prerequisites (i.e., which inputs it needs), and the `calculate()` method *automatically* receives the values of all of its prerequisites as a hash or dictionary at run time.

Figure 1 shows the stubs for three microservices written using Python, Perl, and Java and for a topology defining a toy application using these services. The application takes a student response, tokenizes it into sentences, assigns part-of-speech tags to each sentence, and then computes the similarity between these tagged sentences and a reference answer to the same question for which the student response was written.

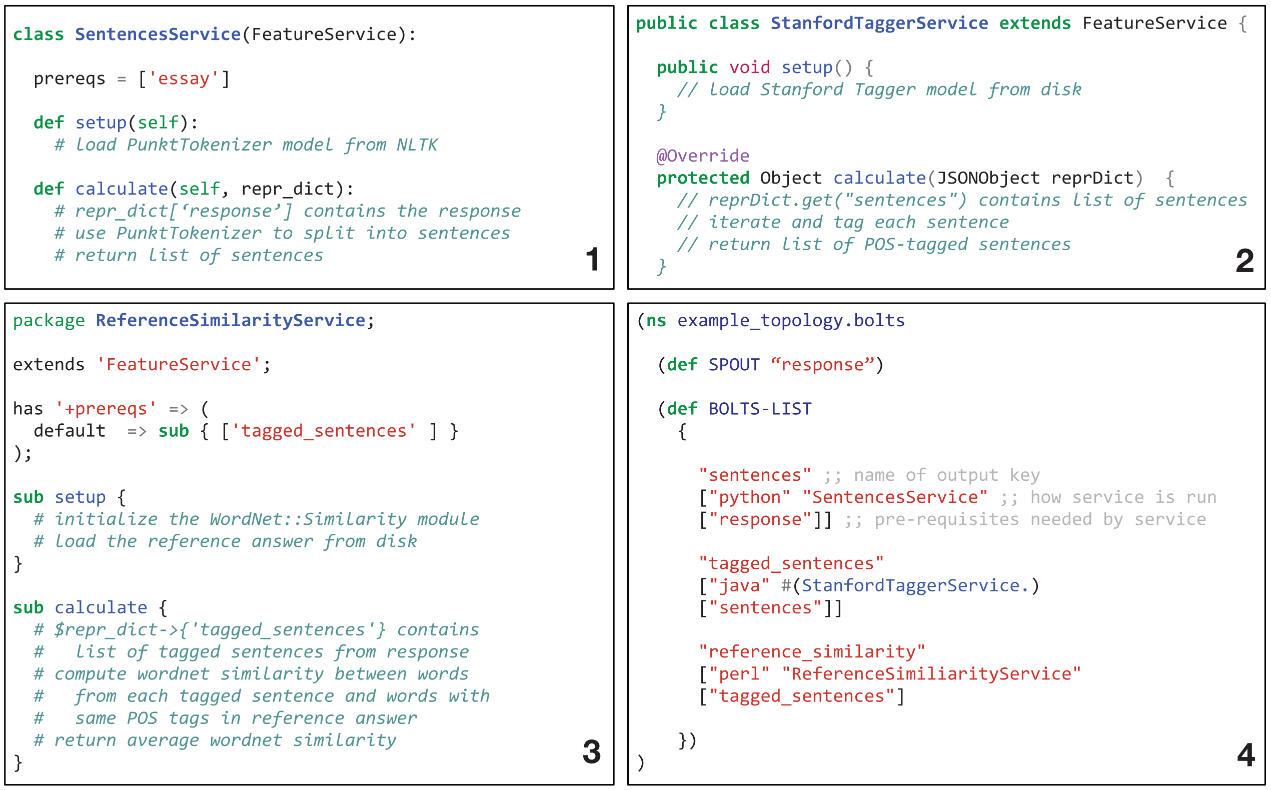
Evaluation

To illustrate the impact of the microservice architecture, we compare the throughput (measured in responses scored per hour) of different versions of our two largest scoring applications—one that scores responses for writing quality using features for discourse structure, vocabulary use, and English conventions (Attali & Burstein, 2006) and a second application that scores responses for understanding of content (Heilman & Madnani, 2015; Madnani, Cahill, & Riordan, 2016). Unlike writing quality, content scoring generally ignores misspellings and grammatical errors (see the Content-Scoring Application section, where we demonstrate the content-scoring application in more detail).

The first version of each application does not use the microservice architecture and is structured as a traditional sequential application. These traditional systems were technically not fully sequential because some parts had been parallelized manually—a significant development effort. The second version uses the microservice architecture with each component implemented as a bolt inheriting from the `FeatureService` class, participating in a predefined topology. This version is able to take full advantage of Storm's automatic parallelization. Table 1 shows their throughput values.[4] For both scoring applications, using the microservice architecture leads to a significant increase in the throughput.

Student responses processed by the writing quality-scoring application are, on average, three to four times longer than the responses processed by the content-scoring application, which leads to lower throughput values for the former in general. In addition, the writing quality-scoring application uses many more features than the content-scoring application, leading to more opportunities for parallelization and, hence, a larger increase in throughput over the traditional version.

Additional evaluation metrics could be applied in this scenario, such as intertask message latency, average task switch time, minimum interrupt latency, and deadlock break time (Gumzej & Halang, 2010, Chapter 2). However, for our purposes, these metrics are superseded by our throughput metrics (which are also the metrics used by IT when running performance tests on the automated scoring engines). In addition, it is difficult to measure effectively some of the other



```
class SentencesService(FeatureService):

  prereqs = ['essay']

  def setup(self):
    # load PunktTokenizer model from NLTK

  def calculate(self, repr_dict):
    # repr_dict['response'] contains the response
    # use PunktTokenizer to split into sentences
    # return list of sentences
```

```
public class StanfordTaggerService extends FeatureService {

  public void setup() {
    // load Stanford Tagger model from disk
  }

  @Override
  protected Object calculate(JSONObject reprDict)  {
    // reprDict.get("sentences") contains list of sentences
    // iterate and tag each sentence
    // return list of POS-tagged sentences
  }
```

```
package ReferenceSimilarityService;

extends 'FeatureService';

has '+prereqs' => (
  default  => sub { ['tagged_sentences' ] }
);

sub setup {
  # initialize the WordNet::Similarity module
  # load the reference answer from disk
}

sub calculate {
  # $repr_dict->{'tagged_sentences'} contains
  #   list of tagged sentences from response
  # compute wordnet similarity between words
  #   from each tagged sentence and words with
  #   same POS tags in reference answer
  # return average wordnet similarity
}
```

```
(ns example_topology.bolts

  (def SPOUT "response")

  (def BOLTS-LIST
    {

      "sentences" ;; name of output key
      ["python" "SentencesService" ;; how service is run
      ["response"]] ;; pre-requisites needed by service

      "tagged_sentences"
      ["java" #(StanfordTaggerService.)
      ["sentences"]]

      "reference_similarity"
      ["perl" "ReferenceSimiliarityService"
      ["tagged_sentences"]

    })
)
```

Figure 1 Illustrating Storm microservices and topologies. Boxes 1, 2, and 3 show stubs for three different `FeatureService` bolts written in Python, Java, and Perl, respectively. Box 4 shows the stub of a topology (written in Clojure) that defines a toy scoring application composed of these bolts.

Table 1 Throughputs for Two Different Scoring Applications (Measured in Student Responses Scored per Hour) for a Traditional Version and for the Version that Uses the Proposed Storm-Based Microservice Architecture

	Version	
	Traditional	μ-Service
Score type		
Quality	6,058	12,091
Content	57,285	70,491

nontangible benefits, such as improved collaborative development, ease of integrating multiple programming languages, and sharing code across engines.

Content-Scoring Application

In this section, we provide more details on how the content-scoring application is architected using the microservice architecture. The application can be tested via a Web app that communicates with the instantiated topology of this application, allowing users to provide answers to several science, math, or English questions and obtaining scores for them in real time.

Scoring responses for writing quality requires measuring whether the student can organize and develop an argument and write fluently with no grammatical errors or misspellings. In contrast, scoring for content deals with responses to open-ended questions designed to test what the student knows, has learned, or can do in a *specific* subject area, such as computer science, math, or biology, without regard to fluency (Dzikovska et al., 2013; Mohler, Bunescu, & Mihalcea, 2011;

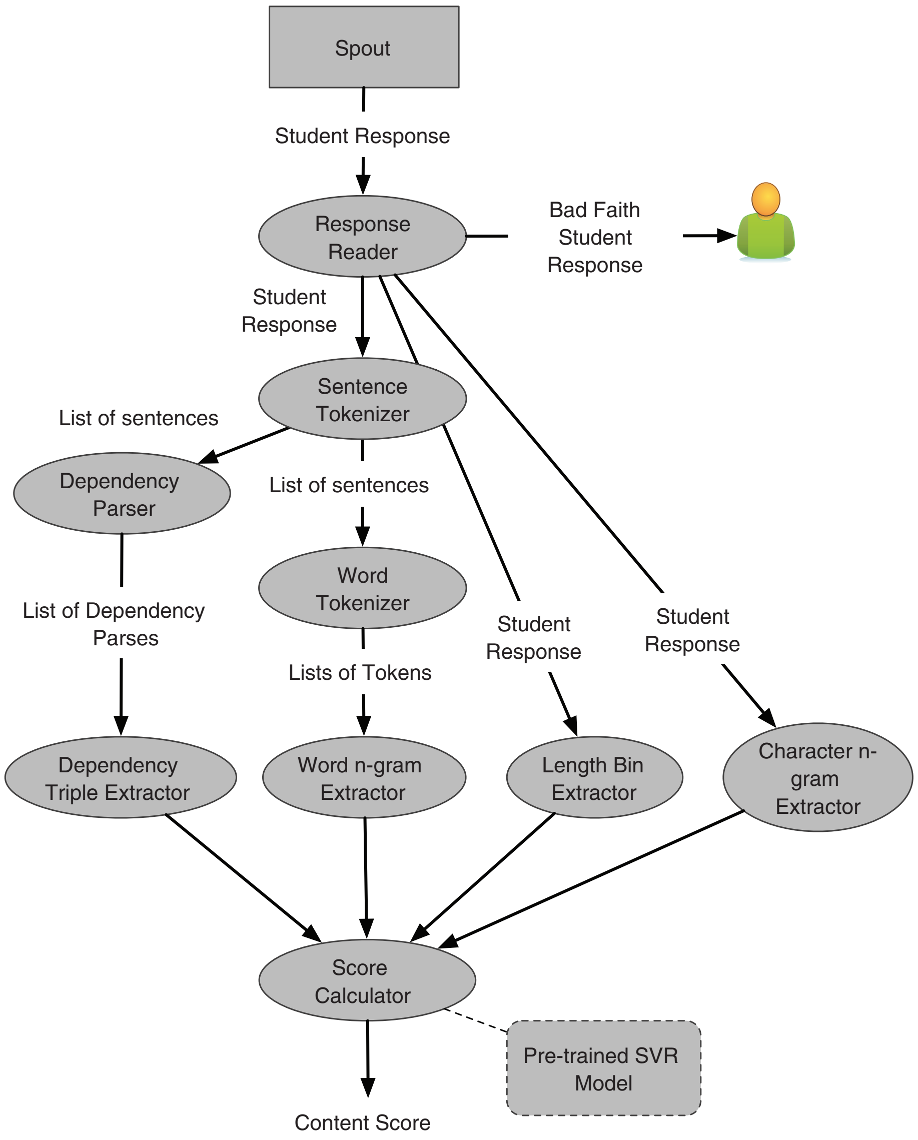
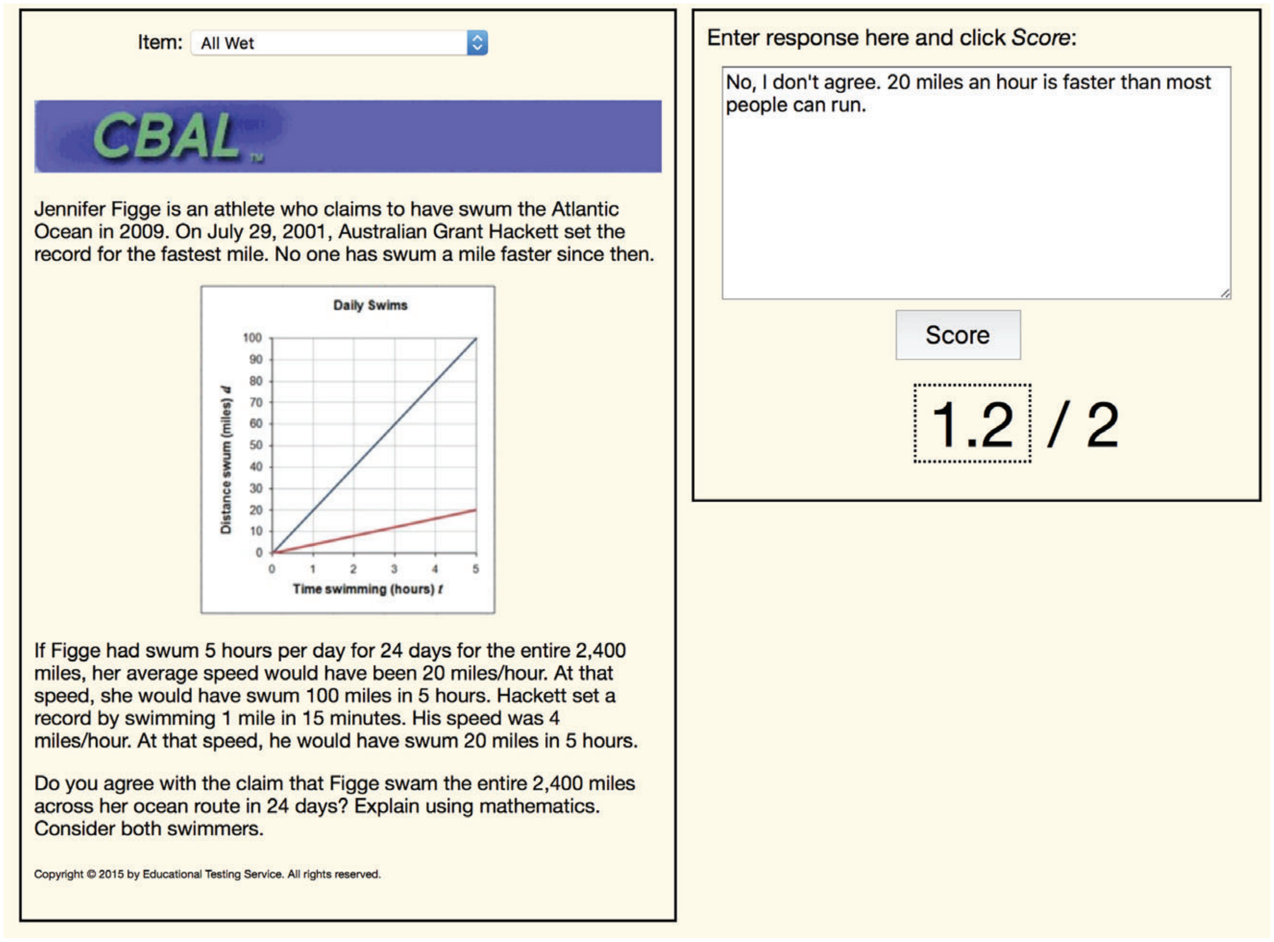


Figure 2 Visualizing the topology of the content-scoring application. Each oval represents a bolt, and each edge between bolts represents a data stream connection. The final bolt uses a pretrained, preloaded support vector regression model to compute the score for a response, based on the four feature streams.

Ramachandran, Cheng, & Foltz, 2015; Sakaguchi, Heilman, & Madnani, 2015; Sukkarieh, 2011; Sukkarieh & Stoyanchev, 2009; Zhu, Liu, Mao, & Pallant, 2016).[5]

The content-scoring application typically uses text regression or classification: We label existing student responses to the question with scores on an ordinal scale (e.g., correct, partially correct, or incorrect; 1–5 score range, etc.), extract a predefined set of features for these responses, and then train a machine learning model with these features using *scikit-learn* (Pedregosa et al., 2011). The trained model is first evaluated for accuracy, fairness, and validity on held-out data (Madnani, Loukina, von Davier, Burstein, & Cahill, 2017) and, if it passes the evaluation criteria, is deployed to production. For our content-scoring application, we use the following set of features:

- lowercased word n-grams ($n = 1, 2$), including punctuation
- lowercased character n-grams ($n = 2, 3, 4, 5$)
- syntactic dependency triples computed using the ZPar parser (Zhang & Clark, 2011)
- length bins (specifically, whether the log of 1 plus the number of characters in the response, rounded down to the nearest integer, equals x, for all possible x from the training set)



Jennifer Figge is an athlete who claims to have swum the Atlantic Ocean in 2009. On July 29, 2001, Australian Grant Hackett set the record for the fastest mile. No one has swum a mile faster since then.

If Figge had swum 5 hours per day for 24 days for the entire 2,400 miles, her average speed would have been 20 miles/hour. At that speed, she would have swum 100 miles in 5 hours. Hackett set a record by swimming 1 mile in 15 minutes. His speed was 4 miles/hour. At that speed, he would have swum 20 miles in 5 hours.

Do you agree with the claim that Figge swam the entire 2,400 miles across her ocean route in 24 days? Explain using mathematics. Consider both swimmers.

Copyright © 2015 by Educational Testing Service. All rights reserved.

No, I don't agree. 20 miles an hour is faster than most people can run.

Figure 3 Screenshot of a Web app communicating with our content-scoring application to score the responses to a math question for which the possible score is in the range 0 – 2.

Figure 2 shows a visualization of the Storm topology of our content-scoring application. Each oval in the figure represents a Storm bolt. There are nine bolts in total. All student responses to be scored enter the topology at the spout and are passed on to the `ResponseReader` bolt, which uses statistical language identification models and a set of simple rules to automatically detect responses that may be written in bad faith, for example, responses in other languages or nonscorable responses of the form “idk,” “I don’t know,” and so on. All such bad-faith responses are directed to a human scorer. Any responses that are not filtered out are then passed to the subsequent bolts that compute intermediate representations (sentences, words, dependency parses) and to bolts that compute the four types of features. Once the features are computed, they are passed to the final bolt, which uses a pretrained support vector regression model to compute a score for the response. Our architecture automatically handles parallelism, for example, while a response is being parsed in the `DependencyParser` bolt, other bolts are processing other responses. It also allows for multiple instances of a bolt, so throughput can be further improved by having multiple instances of slower bolts (e.g., parsers).

Figure 3 shows a screenshot of a Web app that allows a student to enter responses to various content questions. Responses are sent to the topology of the content-scoring application running in the background, and results are presented to the student on the same page. The app contains several questions from different domains (math, science, and English) that can be selected using a drop-down menu. In the screenshot in Figure 3, a math question[6] asks students to evaluate a claim and to explain whether they agree or disagree using their knowledge of mathematics. Students enter their responses in the text box and click “Score” to submit their responses. The score returned by the application is displayed at the top of the screen. In this app, the score is returned in raw format, but other applications might round first before displaying the final score to the student.

Conclusions

Our goal is to provide insights into the rarely discussed topic of transitioning a NLP system, for example, an automated scoring system, from a small-scale research prototype into a production system that can process millions of student responses reliably and with low latency.

We described a robust microservice architecture that can be used to successfully execute such a transition for automated scoring applications. The architecture extends an existing open-source framework that provides several advantages out of the box in terms of scalability, robustness, and fault tolerance. Our extensions to the framework address some of its limitations pertaining to its use for automated scoring applications. We presented concrete examples in which using this architecture leads to significant improvements in throughput processing, ease of development, and scalability. Finally, we showed a Web interface to a microservice-based content-scoring application that can score user-authored responses for content in real time.

Acknowledgments

The authors thank Keelan Evanini, Vikram Ramanarayanan, and Beata Beigman Klebanov for their comments on an earlier draft of the report. We also thank Dmytro Galochkin and John Blackmore for discussions relating to performance evaluation of the microservice architecture.

Notes

1 http://storm.apache.org/.
2 http://github.com/dan-blanchard/io-storm.
3 http://github.com/Parsely/streamparse.
4 The throughput was measured by scoring 250,000 responses for writing quality and 135,000 responses for content. Each application was run on the same single server. The responses were written to questions from a mixture of low-stakes assessments and assessments that aid in making high-stakes decisions.
5 See Table 3 in Burrows et al. (2015) for a detailed list.
6 This question is taken from the *CBAL*® learning and assessment tool (Bennett, 2010).

References

Attali, Y., & Burstein, J. (2006). Automated essay scoring with e-rater v. 2. *Journal of Technology, Learning and Assessment, 4*(3), 1–30.

Bennett, R. E. (2010). Cognitively Based Assessment of, for, and as Learning (CBAL): A preliminary theory of action for summative and formative assessment. *Measurement, 8*, 70–91. https://doi.org/10.1080/15366367.2010.508686

Bird, S., Klein, E., & Loper, E. (2009). *Natural language processing with Python*. Sebastopol, CA: O'Reilly Media Inc.

Burrows, S., Gurevych, I., & Stein, B. (2015). The eras and trends of automatic short answer grading. *International Journal of Artificial Intelligence in Education, 25*, 60–117. https://doi.org/10.1007/s40593-014-0026-8

Burstein, J., & Marcu, D. (2000, August). Benefits of modularity in an automated essay scoring system. In *Proceedings of the COLING-2000 Workshop on Using Toolsets and Architectures to Build NLP Systems* (pp. 44–50). Stroudsburg, PA: Association for Computational Linguistics.

Dong, F., & Zhang, Y. (2016). Automatic features for essay scoring: An empirical study. *Proceedings of EMNLP* (pp. 1072–1077). Stroudsburg, PA: Association for Computational Linguistics.

Dzikovska, M., Nielsen, R., Brew, C., Leacock, C., Giampiccolo, D., Bentivogli, L., … & Dang, H. T. (2013). Task 7: The joint student response analysis and 8th recognizing textual entailment challenge. In *Proceedings of SEMEVAL* (pp. 263–274). Stroudsburg, PA: Association for Computational Linguistics.

Farra, N., Somasundaran, S., & Burstein, J. (2015). Scoring persuasive essays using opinions and their targets. In *Proceedings of the Tenth Workshop on Innovative Use of NLP for Building Educational Applications* (pp. 64–74). Stroudsburg, PA: Association for Computational Linguistics.

Gumzej, R., & Halang, W. A. (2010). *Real-time systems' quality of service*. London, England: Springer. https://doi.org/10.1007/978-1-84882-848-3

Heilman, M., & Madnani, N. (2015). The impact of training data on automated short answer scoring performance. In *Proceedings of the Tenth Workshop on Innovative Use of NLP for Building Educational Applications* (pp. 81–85). Stroudsburg, PA: Association for Computational Linguistics.

Klein, D., & Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of ACL* (pp. 423–430). Stroudsburg, PA: Association for Computational Linguistics.

Liu, O. L., Brew, C., Blackmore, J., Gerard, L., Madhok, J., & Linn, M. C. (2014). Automated scoring of constructed-response science items: Prospects and obstacles. *Educational Measurement: Issues and Practice, 33*, 19–28. https://doi.org/10.1111/emip.12028

Loukina, A., Zechner, K., Chen, L., & Heilman, M. (2015). Feature selection for automated speech scoring. In *Proceedings of the Tenth Workshop on Innovative Use of NLP for Building Educational Applications* (pp. 12–19). Stroudsburg, PA: Association for Computational Linguistics.

Madnani, N., Cahill, A., & Riordan, B. (2016). Automatically scoring tests of proficiency in music instruction. In *Proceedings of the 11th Workshop on Innovative Use of NLP for Building Educational Applications* (pp. 217–222). Stroudsburg, PA: Association for Computational Linguistics.

Madnani, N., Loukina, A., von Davier, A., Burstein, J., & Cahill, A. (2017). Building better open-source tools to support fairness in automated scoring. In *Proceedings of the First Workshop on Ethics in Natural Language Processing* (pp. 41–52). Stroudsburg, PA: Association for Computational Linguistics.

Mohler, M., Bunescu, R., & Mihalcea, R. (2011). Learning to grade short answer questions using semantic similarity measures and dependency graph alignments. In *Proceedings of ACL: HLT* (pp. 752–762). Stroudsburg, PA: Association for Computational Linguistics.

Pedersen, T., Patwardhan, S., & Michelizzi, J. (2004). WordNet::Similarity: Measuring the relatedness of concepts. In *Proceedings of HLT-NAACL (demos)* (pp. 38–41). Stroudsburg, PA: Association for Computational Linguistics.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825–2830.

Ramachandran, L., Cheng, J., & Foltz, P. (2015). *Identifying patterns for short answer scoring using graph-based lexico-semantic text matching*. Paper presented at the Workshop on Innovative Use of NLP for Building Educational Applications, Denver, CO.

Sakaguchi, K., Heilman, M., & Madnani, N. (2015). Effective feature integration for automated short answer scoring. In *Proceedings of NAACL: HLT* (pp. 1049–1054). Stroudsburg, PA: Association for Computational Linguistics.

Shermis, M. D., & Burstein, J. (2013). *Handbook of automated essay evaluation: Current applications and new directions*. New York, NY: Routledge. https://doi.org/10.4324/9780203122761.ch3

Stonebraker, M., Çetintemel, U., & Zdonik, S. (2005). The 8 requirements of real-time stream processing. *SIGMOD Record, 34*(4), 42–47. https://doi.org/10.1145/1107499.1107504

Sukkarieh, J. Z. (2011). Using a MaxEnt classifier for the automatic content scoring of free-text responses. In A. Mohammad-Djafari, J.-F. Bercher, & P. Bessiere (Eds.), *Bayesian inference and maximum entropy methods in science and engineering: Proceedings of the 30th International Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering* (Vol. 1305, pp. 41–48). Melville, NY: AIP Publishing. https://doi.org/10.1063/1.3573647

Sukkarieh, J. Z., & Stoyanchev, S. (2009). Automating model building in c-rater. In *Proceedings of the 2009 Workshop on Applied Textual Inference* (pp. 61–69). Stroudsburg, PA: Association for Computational Linguistics.

Zhang, Y., & Clark, S. (2011). Syntactic processing using the generalized perceptron and beam search. *Computational Linguistics, 37*, 105–151. https://doi.org/10.1162/coli_a_00037

Zhu, M., Liu, O. L., Mao, L., & Pallant, A. (2016, March). *Use of automated scoring and feedback in online interactive earth science tasks*. Paper presented at the 6th IEEE Integrated STEM Education Conference, Princeton, NJ.

Suggested citation

Madnani, N., Cahill, A., Blanchard, D., Andreyev, S., Napolitano, D., Gyawali, B., … Riordan, B. (2018). *A robust microservice architecture for scaling automated scoring applications* (Research Report No. RR-18-14). Princeton, NJ: Educational Testing Service. https://doi.org/10.1002/ets2.12202

Action Editor: Beata Beigman Klebanov

Reviewers: Keelan Evanini and Vikram Ramanarayanan