Towards better data discovery and collection with flow-based programming

Andrei Paleyes
Department of Computer Science
University of Cambridge
ap2169@cam.ac.uk

Christian Cabrera
Department of Computer Science
University of Cambridge
chc79@cam.ac.uk

Neil D. Lawrence
Department of Computer Science
University of Cambridge
ndl21@cam.ac.uk

Abstract

Despite huge successes reported by the field of machine learning, such as voice assistants or self-driving cars, businesses still observe very high failure rate when it comes to deployment of ML in production. We argue that part of the reason is infrastructure that was not designed for data-oriented activities. This paper explores the potential of flow-based programming (FBP) for simplifying data discovery and collection in software systems. We compare FBP with the currently prevalent service-oriented paradigm to assess characteristics of each paradigm in the context of ML deployment. We develop a data processing application, formulating a subsequent ML deployment task, and measuring the impact of the task implementation within both programming paradigms. Our main conclusion is that FBP shows great potential for providing data-centric infrastructural benefits for deployment of ML. Additionally, we provide an insight into the current trend that prioritizes model development over data quality management.

1 Introduction

After achieving considerable success as an academic discipline, machine learning (ML) components are now increasingly deployed in production systems. Government agencies, private companies and individuals all apply ML to solve practical tasks. McKinsey has reported a 25% year to year growth of ML adoption by businesses, with nearly half of respondents reporting revenue increase [Cam et al., 2019].

Deployment of ML typically happens on top of the existing data processing infrastructure. Companies aim to speed up their processing workflows, gain additional insights to aid their decision making, improve detection of anomalous behavior, or provide customers with new functionality based on historical data. ML models are often a centerpiece of such projects. Unfortunately, ML deployment projects face difficult challenges, with companies reporting up to 50% failure rate [Wiggers, 2019]. We believe that one of the reasons for these failures lies in the fact that the majority of ML projects are being deployed on top of existing software solutions which were built to fulfill goals that are important but unrelated to ML, such as high availability, robustness, low latency. However, ML poses a new set of challenges that the majority of existing software architectures are not designed for [Paleyes et al., 2020]. Data processing is one of the areas that cause most concern [Polyzotis et al., 2018], especially in high-scale service-oriented software environments, such as Twitter [Lin and Ryaboy, 2013] or Confluent [Stopford, 2016].

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

As more businesses seek to convert the data they manage into value, it seems reasonable to explore software architectures that could better fit that purpose. In this work we consider the potential of flow-based programming (FBP, Morrison [1994]) as a paradigm for building business applications with ML, and compare it with currently prevalent control-flow paradigms, namely service-oriented architecture (SOA, Perrey and Lycett [2003], O'Reilly [2020]).

There have already been some attempts to enhance SOA with better data handling capabilities [Götz et al., 2018, Gluck, 2020, Safina et al., 2016, Dehghani, 2019]. In our work we wanted to explore radically different approach towards better data management in software systems. So, rather than incrementally improving SOA, we consider FBP due to a range of useful properties that are particular to the paradigm. FBP was created by Morrison [1994], and defines software applications as a set of processes which exchange data via connections that are external to those processes. FBP exhibits "data coupling", which is considered in computing to be the loosest form of coupling between software components. We anticipate that FBP principles can make data-related tasks, such as data discovery and collection, simpler to perform. We illustrate this idea with a simple experiment. We develop an example application separately with each paradigm. We then carry out an ML deployment procedure within both implementations, and analyze how each deployment stage affects the complexity of the codebase. Our conclusions show that while there are a number of trade-offs to consider, FBP has potential to simplify deployment of ML in data-driven applications.

Data flow paradigms are not new in software engineering, a duality of control flow and data flow for building software has long been explored by the computer science community [Treleaven, 1982, Lauer and Needham, 1979, Hasselbring et al., 2021]. FBP's potential to improve software quality and maintenance has been shown in comparison with other paradigms and design principles, such as OOP [Morrison, 2010], functional programming Roosta [2000], and SOA in IoT context [Lobunets and Krylovskiy, 2014]. But to our knowledge FBP has never been compared to SOA in the context of ML deployment before. Some of the high level ideas that motivated this paper were first introduced by Diethe et al. [2019] and further developed by Lawrence [2019] and Borchert [2020] under the name of Data Oriented Architectures (DOA). Our work can be seen as the first step towards applying DOA to practical tasks.

2 Experiment setup

To explore the different software paradigms we developed an example application to study properties of FBP and compare it against the more widespread SOA approach. Concretely, we implemented a prototype of a taxi ride allocation system described by Lawrence [2019]. The application receives data about currently available drivers and incoming ride requests, and outputs the allocated rides. The application also processes updates of each allocated ride, and keeps track of factual passenger wait times. We formulated a business problem that can be solved with ML: provide user with an estimated wait time, in addition to the allocated driver. Training data for the ML model can be collected based on historical wait times. This type of additional functionality has been shown to be among the major contributors to project's technical debt [Molnar and Motogna, 2020]. As a result we focus our evaluation on changes in code quality.

Two separate implementations of the described application were created: one with FBP using flowpipe[1] and one with SOA using Flask[2]. Detailed description of the application, full source code and list of metrics used for evaluation can be found at `https://github.com/mlatcl/fbp-vs-soa/tree/ride-allocation`.

In order to enable structured approach towards evaluation of codebase changes over the course of ML deployment, we defined three stages of the implementation:

- **Stage 1**: minimal code to provide basic functionality. The stage is denoted in the code and this paper by suffix *min*.
- **Stage 2**: same as Stage 1 plus dataset collection. A complete dataset required collecting data from two locations within the application. Inputs, which we considered to be ride requests and driver locations, are available at the time ride allocation is done. Output, which is the

[1]flowpipe is available at https://github.com/PaulSchweizer/flowpipe. It is considered to be an FBP-inspired framework, but provides all FBP features critical for our work and is easy to read and understand.

[2]https://flask.palletsprojects.com/

Table 1: List of all created versions of the Ride Allocation app. First column gives the key by which a particular version is referred to in the codebase.

Key	**Paradigm**	**Stage**	**Description**
fb_app_min	FBP	1	Basic functionality
fb_app_data	FBP	2	Same as fb_app_min plus dataset collection
fb_app_ml	FBP	3	Same as fb_app_data plus estimated wait time output
soa_app_min	SOA	1	Basic functionality
soa_app_data	SOA	2	Same as soa_app_min plus dataset collection
soa_app_ml	SOA	3	Same as soa_app_data plus estimated wait time output

actual waiting time, becomes available later in different part of the app, when passenger pickup happens. Denoted by suffix *data*.

- **Stage 3**: same as Stage 2 plus the new output of estimated wait time produced via a deployed ML model. The ML model is trained on the dataset collected at the previous stage. The application has to load an already trained offline and serialized ML model, perform the prediction at the time ride allocation is done, and output estimated wait time in addition to the allocation information. Denoted by suffix *ml*.

Overall we ended up with 6 versions of the Ride Allocation system, which are listed in Table 1.

We use a number of software metrics to assess the impact of each subsequent stage on the overall quality of the codebase. These metrics measure size, complexity and maintainability of the code. This metric-based evaluation approach was chosen to enable objective evaluation of codebase quality and how the ML deployment process affected it at each stage.

3 Experiment Results

In this section we analyze and discuss our observations from each stage of the experiment.

Size and complexity metrics and our own observations suggest that initial cost of developing the FBP solutions is higher. That is likely the consequence of the fact that SOA is a highly evolved and widely deployed programming paradigm. Therefore the majority of people with experience of modern industrial software development, which authors of this work consider themselves to be, can iterate and make progress within this paradigm at a quicker pace. In contrast, FBP is not nearly as widespread. This paradigm requires a conceptual shift in the way a developer thinks about the application, because instead of customary control-flow one needs to adopt data-flow mindset. However, cognitive complexity metric suggest that FBP programs are easier to read and comprehend once they are written (see Figure 1).

Dataset collection stage turned out to be the most critical for surfacing differences between the paradigms in the ML deployment context. FBP programs allow programmatic access to the whole dataflow graph, with nodes representing business logic of the application, and edges representing flows of data, making it possible to programmatically access data flowing to or from any node. Thus changes were made in single location of the codebase, even though we needed to collect data from multiple internal sources. In contrast, changes to the SOA application had to be introduced in multiple places, which impairs code readability and long term support.

Unlike the previous two stages, the model hosting stage yielded no additional insight into difference between the paradigms considered. Nevertheless it is important to see the confirmation of

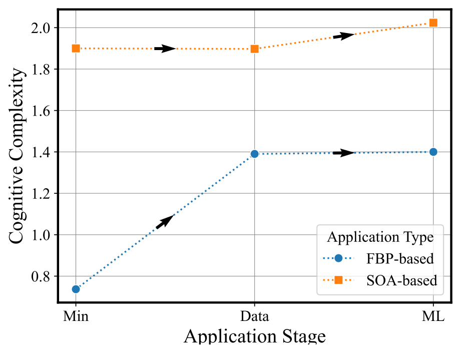
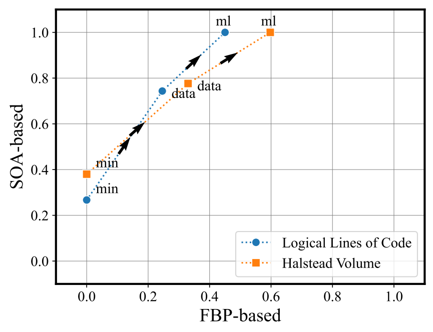


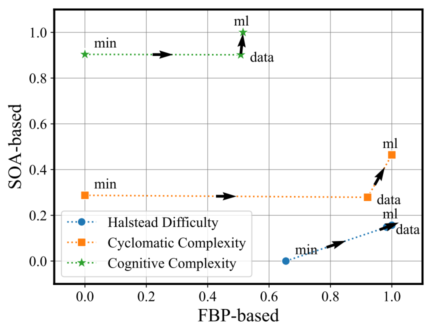
Figure 1: Cognitive complexity metric measured on both implementations of the Ride Allocation app. Higher value means code is more complex.

the fact that both paradigms can support hosting ML model for predictions without significant impact on the rest of the system.

Comparing behavior of multiple code complexity metrics we have realized that the data collection stage was far more impactful change than the model deployment (Figure 2). This might uncover additional reason for the trend in modern ML community to focus on model research rather then data research [Lawrence, 2017]. If making changes to deployed model is easier and less error-prone than making changes to data engineering pipeline, it is easy to understand why developers and researches are motivated to seek improvements in model iterations rather than over data quality. Nevertheless we believe data management is equally important part of machine learning process, especially since data scientists spend most of their time working with data [Nazabal et al., 2020]. With this work we aim to make a step towards simplifying data-oriented tasks in software systems.



(a) Code size metrics combined. The trends intersect between stages *min* and *data*, while being nearly parallel between stages *data* and *ml*. This suggests that data collection on *data* stage had different scale of impact on metrics for FBP and SOA implementations.



(b) Code complexity metrics combined. The distance between points that represent stages *min* and *data* is significantly bigger than between points for stages *data* and *ml*. This shows how big was an impact on code complexity by the implementation of data collection.

Figure 2: Combinations of multiple code metrics. All values are normalized to fall within $[0, 1]$ range.

4 Conclusions and Future Work

In this paper we illustrated the potential of using FBP to ease the pain of deploying ML and improve data management. In a software system designed according to FBP principles the tasks of data discovery and collection become more straightforward, thus simplifying consequent deployment of ML. We believe better tooling that allows developers to define dataflow graphs at a higher level of abstraction would help fill some of the current gaps and leverage that potential.

Additionally, we showed that data collection code caused much more significant impact to metrics of both codebases, compared to model deployment. This could be an explanation of modern trends of seeking performance improvements through models rather than through data.

We observed that when developing an application with FBP paradigm, a lot of effort is spent in defining and manipulating the dataflow graph. On the other hand, once such a graph is defined, all data flows in the system become explicit, thus making data discovery task simpler. Any framework that allows software developers to abstract away from the boilerplate code and focus on actual application domain, business logic and entities, would streamline the development process and reduce the complexity of the codebase. There are tools that might serve this purpose, such as Google Dataflow [Krishnan and Gonzalez, 2015], Kubeflow [Bisong, 2019] and Apache NiFi[3], although they are usually seen as very specific to particular applications. Understanding commonalities of these frameworks is a promising starting point for building general purpose development tools.

[3]https://nifi.apache.org/

In the future we would like to further scale the experiment described in this paper. For instance, the same ML deployment perspective can be considered in the distributed context, where data streaming platforms such as Apache Kafka would have to be used. Long-term experiments can also be informative to observe the code evolution over a longer period, e.g. a year. Other paradigms, such as the Actor model [Hewitt, 2010], might be considered for comparison.

Acknowledgments and Disclosure of Funding

We would like to thank our colleagues Pierre Thodoroff, Markus Kaiser, Eric Meissner, Jessica Montgomery, Diana Robinson for many insightful discussions.

References

Arif Cam, Michael Chui, and Bryce Hall. Global AI survey: AI proves its worth, but few scale impact. *McKinsey Analytics*, 2019.

Kyle Wiggers. IDC: For 1 in 4 companies, half of all AI projects fail, 2019. Available at `https://venturebeat.com/2019/07/08/idc-for-1-in-4-companies-half-of-all-ai-projects-fail/`.

Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. Challenges in deploying machine learning: a survey of case studies. *arXiv preprint arXiv:2011.09926*, 2020.

Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data lifecycle challenges in production machine learning: a survey. *ACM SIGMOD Record*, 47(2):17–28, 2018.

Jimmy Lin and Dmitriy Ryaboy. Scaling big data mining infrastructure: the Twitter experience. *ACM SIGKDD Explorations Newsletter*, 14(2):6–19, 2013.

Ben Stopford. The data dichotomy: Rethinking the way we treat data and services, 2016. Available at `https://www.confluent.io/blog/data-dichotomy-rethinking-the-way-we-treat-data-and-services/`.

J. Paul Morrison. Flow-based programming. In *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 25–29, 1994.

Randall Perrey and Mark Lycett. Service-oriented architecture. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.*, pages 116–119. IEEE, 2003.

O'Reilly. Microservices adoption in 2020: A survey, 2020. Available at `https://www.oreilly.com/radar/microservices-adoption-in-2020/`.

Benjamin Götz, Daniel Schel, Dennis Bauer, Christian Henkel, Peter Einberger, and Thomas Bauernhansl. Challenges of production microservices. *Procedia CIRP*, 67:167–172, 2018.

Adam Gluck. Introducing domain-oriented microservice architecture. *Uber Engineering Blog*, 2020.

Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices: Genericity in jolie. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 430–437. IEEE, 2016.

Zhamak Dehghani. How to move beyond a monolithic data lake to a distributed data mesh. *Martin Fowler's Blog*, 2019.

Philip C. Treleaven. Towards a decentralised general-purpose computer. In *Programmiersprachen und Programmentwicklung*, pages 21–31. Springer, 1982.

Hugh C Lauer and Roger M. Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, 1979.

Wilhelm Hasselbring, Maik Wojcieszak, and Schahram Dustdar. Control flow versus data flow in distributed systems integration: Revival of flow-based programming for the industrial internet of things. *IEEE Internet Computing*, 25(4):5–12, 2021.

J. Paul Morrison. *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010.

Seyed H. Roosta. Data flow and functional programming. In *Parallel Processing and Parallel Algorithms*, pages 411–437. Springer, 2000.

O. Lobunets and A. Krylovskiy. Applying flow-based programming methodology to data-driven applications development for smart environments, 2014.

Tom Diethe, Tom Borchert, Eno Thereska, Borja Balle, and Neil D. Lawrence. Continual learning in practice. *arXiv preprint arXiv:1903.05202*, 2019.

Neil D. Lawrence. Modern data oriented programming, 2019. Available at `http://inverseprobability.com/talks/notes/modern-data-oriented-programming.html`.

Tom Borchert. Milan: An evolution of data-oriented programming, 2020. Available at `https://tborchertblog.wordpress.com/2020/02/13/28/`.

Arthur-Jozsef Molnar and Simona Motogna. Long-term evaluation of technical debt in open-source software. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–9, 2020.

Neil D. Lawrence. Data readiness levels. *arXiv preprint arXiv:1705.02245*, 2017.

Alfredo Nazabal, Christopher K. I. Williams, Giovanni Colavizza, Camila Rangel Smith, and Angus Williams. Data engineering for data analytics: a classification of the issues, and case studies. *arXiv preprint arXiv:2004.12929*, 2020.

S. P. T. Krishnan and Jose L. Ugia Gonzalez. Google Cloud Dataflow. In *Building Your Next Big Thing with Google Cloud Platform*, pages 255–275. Springer, 2015.

Ekaba Bisong. Kubeflow and kubeflow pipelines. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 671–685. Springer, 2019.

Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.