SAIL: A Scalable Architecture for Inference and Learning

White Paper

TAMÁS FÖLDI

CTO/Partner, USA

CHRIS VON CSEFALVAY

VP Special Projects

Contents

Starschema, Inc. 2221 South Clark St., Arlington, VA 22202 +1 415 944-5455

Introduction

This paper discusses a paradigm of machine learning automation that builds greatly on well-known DevOps tools that have seen little use in the data science, machine learning and analytics field. As we witness the emergence of DataOps, a discipline of ML engineering focused on leveraging DevOps tools for data science and machine learning, new avenues will continue to open up for faster, more efficient and more consistent data processing. With fierce competition and increasing reliance in the field, much depends on a sound methodology for bringing DevOps to the data science arena.

In this document, we introduce an architecture that leverages best practices for the rapid, scalable deployment of data science, machine learning and AI applications that have recently emerged in three fields belonging to DevOps:

- test and behavior driven development (TDD and BDD, respectively);
- continuous integration and continuous deployment (CI/CD); and
- infrastructure and process orchestration (Kubernetes and Docker).

While these have become ubiquitous tools within modern application development, their immense utility in the data science field has gone largely unrecognized. Yet many of the resource-intensive parts of data science and ML engineering can be automated and accelerated using these technologies, while also retaining constant introspection into model accuracy and performance.

The Standard Approach to Inference and Learning (SAIL) is a fast, efficient and scalable way to facilitate rapid, efficient and agile machine learning as well as inference. Rather than opinionated prescriptive architectures, a good SAIL is not limited to any particular language, machine learning framework or machine learning approach, such as neural networks. Moreover, it is not solely limited to machine learning, but also accommodates data processing, transformation and mapping tasks commonly performed by ETL.

Designed to answer the twin demands of consistency and replicability, SAIL uses Docker containers to create loosely coupled parts of a data transformation, machine learning and inference pathway' and uses CI/CD pipelines to enable a process we refer to as Continuous Learning: the incremental refinement of models based on new data and changing hyperparameters, continuously tracked in Git and Docker image repositories. This ensures auditable workflows, the ability to easily roll back changes and even to select between different models to accommodate various use cases.

Starschema, Inc. 2221 South Clark St., Arlington, VA 22202 +1 415 944-5455

Theoretical foundations

The workflow of data science and machine learning tends to be divided into two parts. Training fits a model over a part of the input data set — known as the training set. In this context, testing —inference applied to a part of the input data set that has not been used to train the model — is considered part of training. Finally, the trained model is serialized — abstracted to a format that can be committed to durable storage. For a neural network, this may consist of 'dumping' the weights of the network and its architecture as a JSON file. While for a simple linear regression, it may consist of the coefficients of regression. Inference, on the other hand, focuses on new incoming data based on a pretrained model.

Of course, this picture omits several important aspects, such as quality assurance (QA) of models, but it outlines the main distinction between training-side and inference-side work. Training is almost always carried out in batches and is relatively rarely re-run. This differs from inference, which may be carried out in batches or, if appropriate, served as an API.

SAIL is premised on the 'separation where needed, sharing where possible' principle. Separating inference from training and testing reflects a fundamental rule of data science that inference should be over new data. However, sharing a common transformation layer ensures that the learning algorithm is trained and tested over data subjected to the same transformations that incoming source data will be subjected to, and results in a high level of consistency and replicability.

The integration of model quality introspection and QA is an essential part of the SAIL workflow. In software development, CI/CD workflows have, since their inception, integrated automated testing tools, such as behavioral, unit and even end-to-end testing that looks at the DOM rendered by a web application (e.g. using Selenium). Continuous Learning, the paradigm at the very heart of SAIL, adapts the underlying idea of continuous improvement through testing and quality metrics to the machine learning context.

By setting either a minimum performance for models — expressed e.g. as the ROC AUC over the test set, the F1 metric or some other quality indicator — or putting models into production only where their performance exceeds the previous model's over similar metrics (regression testing), we can ensure that even without manual approval, reliable models are automatically trained and passed onto the inference side. This may be augmented by comparative metrics of input data — in general, input data does not radically change overnight, and sudden large changes of input data statistics by variable indicate a data quality error.

Simple methods of anomaly detection, such as first derivative based limits or exceeding maxima or minima by a certain margin, are often sufficient to cater for the vast majority of

data quality issues. Where reliability of input data is critical, such as in safety-critical applications, more complex methods of anomaly detection and aggressive limits on outliers can be imposed to reject inconsistent data.

SAIL in practice

The SAIL architecture reflects the separation of concerns in data science and machine learning applications into training and inference by the two different 'sides' of the SAIL architecture, connected by the shared components.

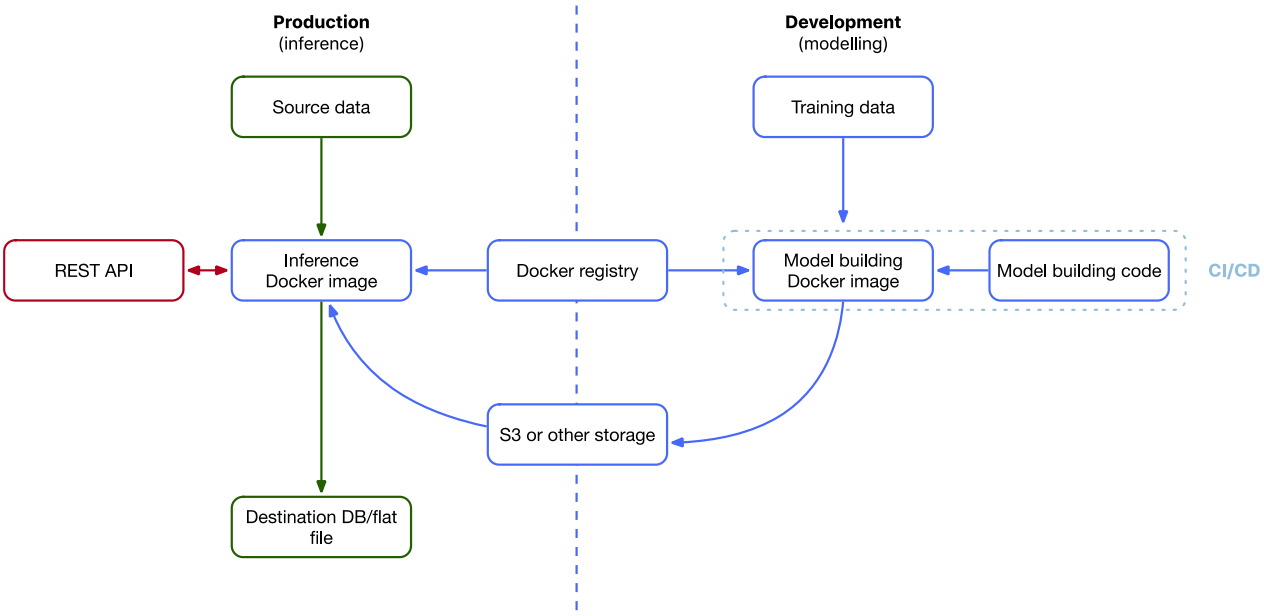

Figure 1. Schematic overview of the SAIL architecture. Based on model building code, a Docker image is built and instantiated as a container, run through a CI/CD pipeline. The model ingests training data and outputs a model to the bulk file storage. The inference image is instantiated in another container and draws on the model in the bulk file storage to perform the inference task.

The training side (right) begins with the training data set, typically stored in a database or a suitable file format, such as a CSV file or, for more complex data, HDF5. This data is passed through the Shared Transformation Layer (STL), which imposes the same shared transformations on training data, test data and data used in inference, thereby ensuring consistency of inputs. On the training side, the STL's output is passed to the model training layer, which returns a serialized model as well as metadata, including model performance over a test set — pre-defined or randomly selected as a percentage of the input data. The model is then serialized and, along with its metadata, saved in bulk file storage, such as AWS's S3.

The inference side ingests data from a source, typically either an API call's input or data from a database when run as a batch job. Upon ingestion, the data is passed through the

STL and then passed onto the inference layer. The inference layer is a standard layer specific only to the kind of model being used. Its primary task is to select a suitable model — typically, the most recent model in the bulk file storage that meets the requirements — load it and manage inference. This layer may be relatively short-lived, e.g. for a batch job, or somewhat more enduring, such as where the model is served as an API and the inference layer stands prepared to receive API calls and return near-real-time results.

In the reference implementation, two CI/CD pipelines — a training pipeline and an inference pipeline — are responsible for process orchestration. This forces a degree of consistency on both sides, as the same initial transformations are made on training, test and production data sets. In training, the CI/CD pipeline executes the training process, evaluates the result by running the resulting model over a test set and — if the model has passed evaluation criteria, such as minimum ROC AUC and non-regression — deposits the serialized model in a simple file storage like S3. This can either be a scheduled process or triggered by large data uploads. AWS in particular is suitable for building an infrastructure that supports cross-service calls, but any microservice-driven architecture can be configured to operate in this manner.

The inference pipeline, on the other hand, is run much more frequently. In general, the inference pipeline's main purpose is to instantiate a standard image with all that is required to run inference tasks — i.e. a consistent inference environment — retrieve the latest serialized model and rehydrate (deserialize) it. This model can then be served from a lightweight REST API shell — performing inference on the input in near-real-time — or used to perform batch inference, e.g. on data in an RDBMS table, to be output to a results table or a flat file format such as CSV or JSON.

The two pipelines touch on two important points: the Docker image repository ensures that the pre-processing images are consistent and the images themselves are versioned, and the model store keeps track of serialized models along with metadata. Gold standard implementations should, on the inference side, ingest not only models but also their metadata, allowing users to see the diagnostic information of the generating model — such as ROC AUC — alongside inference from the model. This may be relevant where outputs from multiple models need to be weighted or where users must be provided confidence scores for audit purposes.

Starschema, Inc. 2221 South Clark St., Arlington, VA 22202 +1 415 944-5455

Reference implementation

In a reference batch inference, implementation is disclosed building on AWS CodePipeline as the CI/CD solution, AWS ECR as the Docker container registry and S3 as bulk file storage. Note that certain aspects of deployment — in particular, deployment of the Docker images using Fargate or EC2 instances via ECS, as well as the detailed working of AWS CodePipeline — is obscured for simplicity's sake.

This is an example of a possible SAIL implementation and many other alternatives including hybrid cloud solutions, leveraging Jenkins or other CI/CD tools. Equally, the codebase can be stored in a range of distributed source code management systems, and a number of other solutions and platforms can be used to support the SAIL architecture. Indeed, one of the strengths of the SAIL concept and Continuous Learning is that it is not an opinionated set of practices or tools but rather a distillation of best practices in operation.

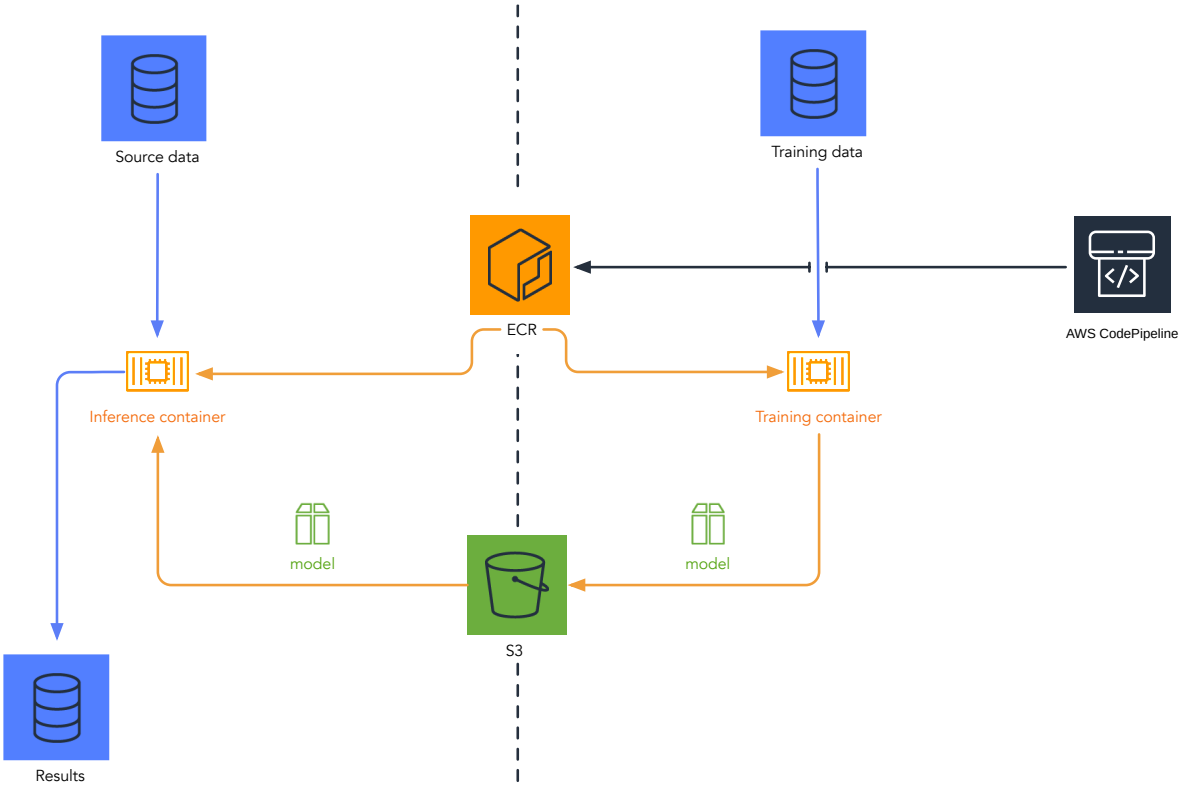

Figure 2. Reference implementation using AWS CodePipeline, AWS Elastic Container Registry (ECR) and S3 for bulk file storage. For simplicity's sake, the detailed workings of the CI/CD workflow in AWS CodePipeline are not shown individually.

Training-side operations

Most of the training-side operations are performed by the training container (TC). The TC is instantiated based on an image in ECR, which is typically built as the end result of a successful AWS CodePipeline run: as code is committed to a source code repository — in this case, this would typically be AWS CodeCommit — a new image is built via preset build parameters in AWS CodeBuild, part of the AWS CodePipeline suite. If this image passes certain pre-set tests and conditions, it is then pushed to the ECR registry, a managed Docker container registry. AWS CodeDeploy then uses this image to raise the TC, either via Fargate or via EC2 deployment. The latter is often more suitable where direct access to the training container may be necessary. Typically, the TC would store logs in a separate S3 logging bucket, which can then be inspected using the ELK stack (Elasticsearch, Logstash and Kibana).

Once the training container has built a model and run model diagnostics, it will compare these to pre-set criteria, such as ROC AUC value, cut-offs for accuracy/recall metrics or other metrics like the F1 score or the Matthews correlation coefficient. If the model is suitable, it will be stored in S3. It is a good idea to store the model alongside some metadata, including metadata to identify what state of training data the model was built from and what its estimated accuracy is. Especially in applications where models may be ensembled to form a meta-model or where they are used in critical decision-making, such as medical imaging, high-frequency/algorithmic trading or Safety Critical Applications, it is important to understand the accuracy of a model and what data it is based on. Often, in such applications, pre-training assay of data quality — e.g. testing for the Newcomb-Benford law, distribution fitting and autocorrelative anomaly detection — might be useful to avoid the problem of 'garbage in, garbage out'.

In general, model serialization and deserialization are catered for by the training container, specifically, by the model package. This may be as simple as storing the coefficients of regression for a simple linear regressor or as complex as training a neural network and separately storing the architecture and the individual neuron weights. While no universal 'silver bullet' exists for all models, Predictive Model Markup Language (PMML) is a widely supported XML-based serialization format for model portability that is often a good choice. Similarly, many of the mainstream Python packages used in modelling, such as scikit-learn, create easily serializable objects that support Python's own persistence model — mainly through the pickle package.

Finally, where this is necessary or appropriate, developers may opt to build their own serialization methods — this may be useful where the ability to compare individual coefficients within the model is desirable as such features are obscured when a binary serialization like pickle is employed. SAIL supports any of these approaches and can be

adjusted to accommodate either. In addition, SAIL can be used to manage the insecurity of many serialization formats — including pickle — by storing a signature of the original model in the metadata and only allowing inference to be made if the metadata and the pickled object's HMAC hashes agree. This is a crucial feature to avoid model tampering or accidental model corruption.

Inference-side operations

The inference side's workhorse, the inference container (IC), is typically much more generic than the training image. In a batch processing use case, the IC loads and deserializes the latest compliant model from S3 and performs a batch inference on the source data, saving the results in a flat file or a database. Where the serialization method used is not secure, an inference image should preferably also handle authentication of the model, confirming the model against a HMAC hash or other hash algorithm.

Frequently, the inference-side operation is triggered by changes in the source data or in the latest model. AWS provides serverless functions called Lambdas to respond to simple triggers and Step Functions to perform more complex orchestration of microservices. With Step Functions and AWS Lambda, a wide range of triggers can be used to determine

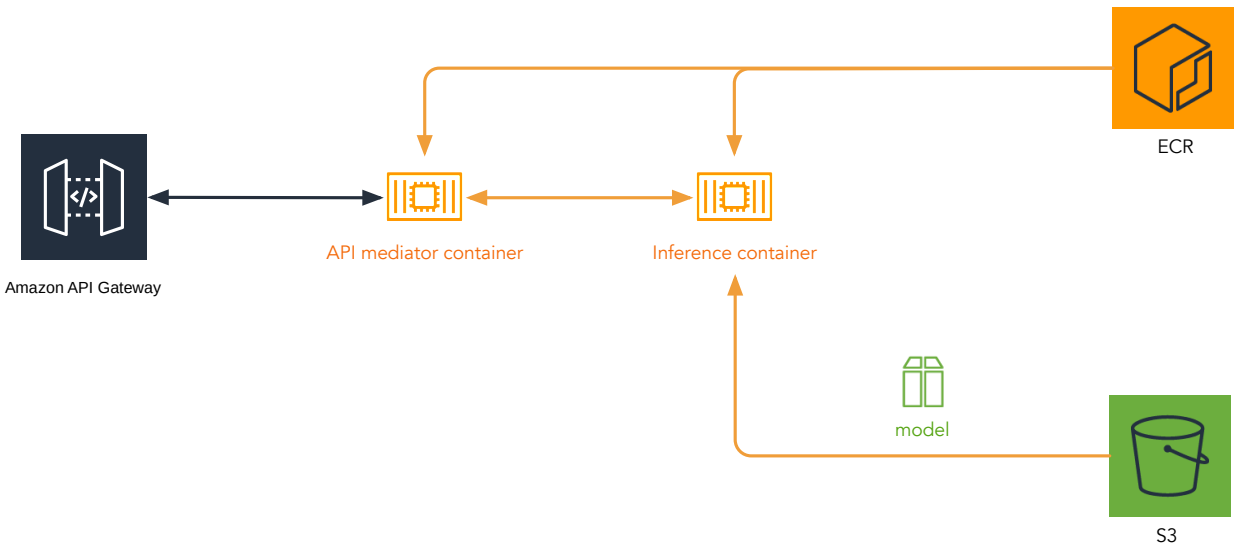

Figure 3. Using Amazon API Gateway and an API mediator container, inference containers can be exposed as RESTful or Websocket APIs to consumers as diverse as mobile, desktop and web applications.

when a model will be run. The ideal frequency of re-inferencing depends on the extent of change — for instance, it may not be economical to perform inference every time a fast-changing database changes a single value, but after a given number of changes, the source data can be considered sufficiently different to warrant re-inferencing.

Serving results as an API

RESTful APIs have become the lingua franca of near-real-time applications. Where batch inference is not necessary or warranted, exposing the inference container via an API is often useful. This is the case, in particular, where inference is relatively computationally inexpensive and will only be required for a relatively small subset of data — often utilization of data in an API obeys the Pareto distribution — approximately 80% of all requests are directed at the same 20% of targets — and in those cases, on-the-fly inference via an API can reduce costs without significantly affecting performance when compared to batch inference.

By using Amazon API Gateway, an API mediator container can expose the inference container's capabilities as a RESTful API or even a Websocket API. Together with a SAML 2.0 provider like Amazon Cognito, it can serve as a fully managed API endpoint/service with identity and access control features, while preserving the dynamic nature of Continuous Learning: the inference container can be updated independently from the API mediator container to use the most recent valid version of the model. This opens up access to the IC — and thus to the model — to a wide range of access modalities.

Once deployed, the API can be accessed from other services, mobile applications, web frontends — including real-time Websockets connections — and desktop applications, and enable real-time inference to be a part of a microservice-based Service Oriented Architecture (SOA).

Advanced topics

This section discusses advanced features of the SAIL architecture. While the core architecture is almost infinitely expandable to suit most use cases, there are some common issues that arise in implementations. Some of these recurrent issues are discussed below.

Jenkins and Github integration

In many practical applications of the SAIL paradigm, it is common to leverage Github or a similar Git-based distributed source code management system host as the code repository and Jenkins as the CI/CD runner. In these cases, Jenkins is typically integrated with Github to build the Docker image on commit and, if the build is successful, upload it to ECR. A separate Jenkins job — which may or may not be on the same server — then runs the training container and deposits the model into S3.

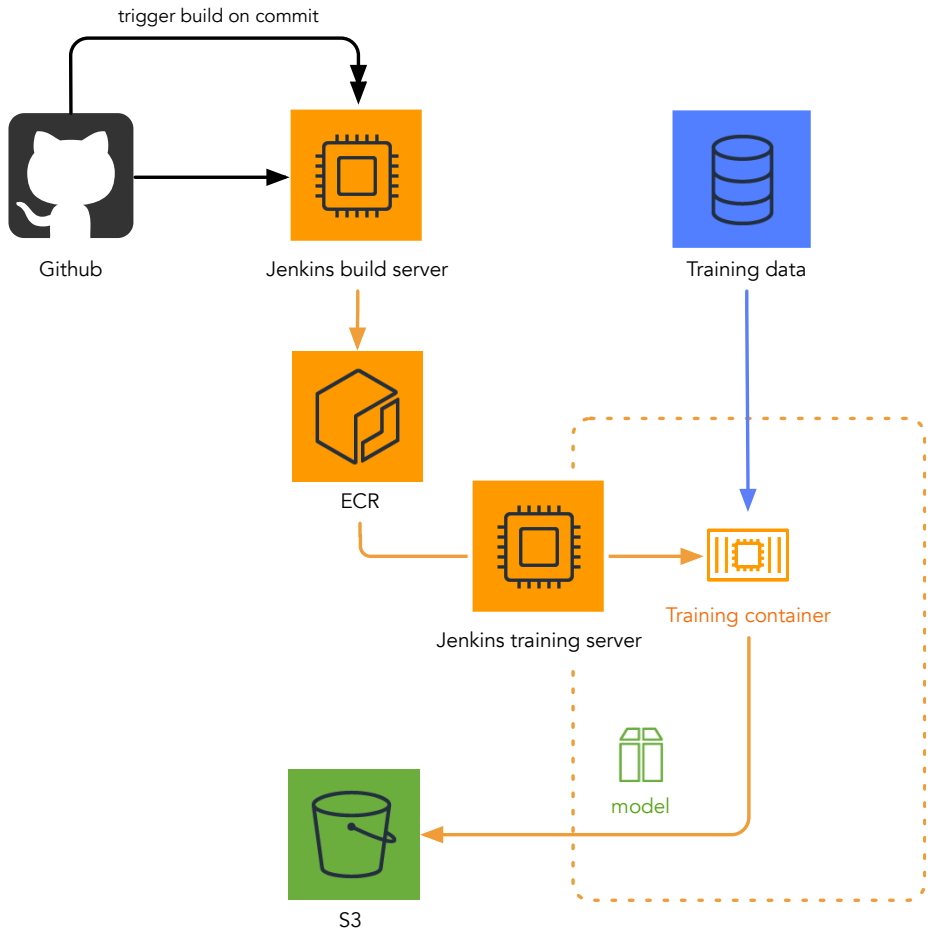

Figure 4. SAIL workflow using Jenkins as the CI/CD engine. A Jenkins build server creates Docker images on commit and stores them in ECR. The training server manages the training pipeline using the training container, instantiated from the image in ECR.

Model performance and testing

One of the virtues of CI/CD that has made it a mainstay of DevOps workflows is the ability to test code using unit testing, behavioral testing or indeed any other arbitrary testing framework. The SAIL architecture allows you to leverage the same flexibility with statistical metrics of model performance.

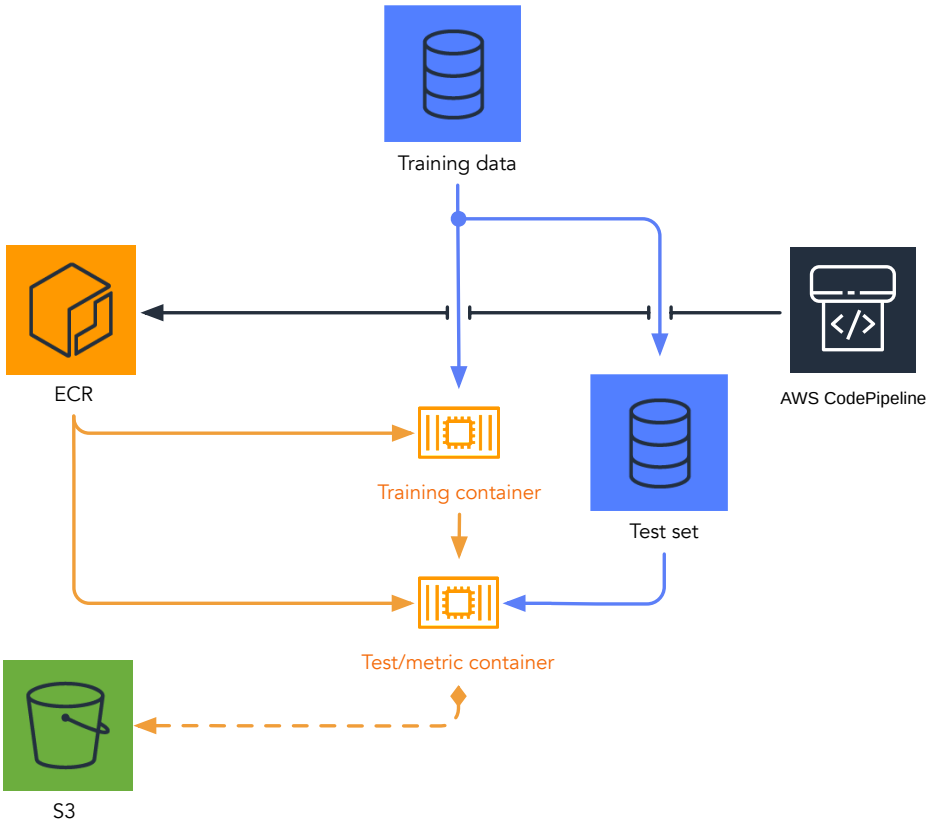

Figure 5. Model introspection using a separate container.

Model introspection and accuracy measurements can be integrated in the training container, in which case a build resulting in a model of subpar accuracy would be considered a run failure and not result in the faulty model being saved to the model repository, e.g. S3.

Alternatively, for more complex measurements, a separate container can be used. This also preserves auditability and isolates the training from the test phase. The training data is physically separated into a test set and a training set before it is passed to the training container, and a separate test/metric container (see) runs performance metrics on the model emerging from the training container using the test/metric container. Typically, this can be used not only to determine whether model accuracy exceeds a particular metric but also used to optimize hyperparameters for non-regression — i.e. only committing models that perform equally well or better than all other models over the same data set.

Multi-model training

A strength of the SAIL architecture is that it also allows for rapid comparison of multiple models. Multi-container parallel training (see) achieves this by using separate containers for each modelling approach. This is warranted, for example, where each of the model types will additionally be hyperparameter optimized. Parallel training is also faster, saving valuable time when compared with serial training in a single container. The test/metric container then determines, based on the test set, which model performed best and stores that model in the bulk file storage (S3 in).

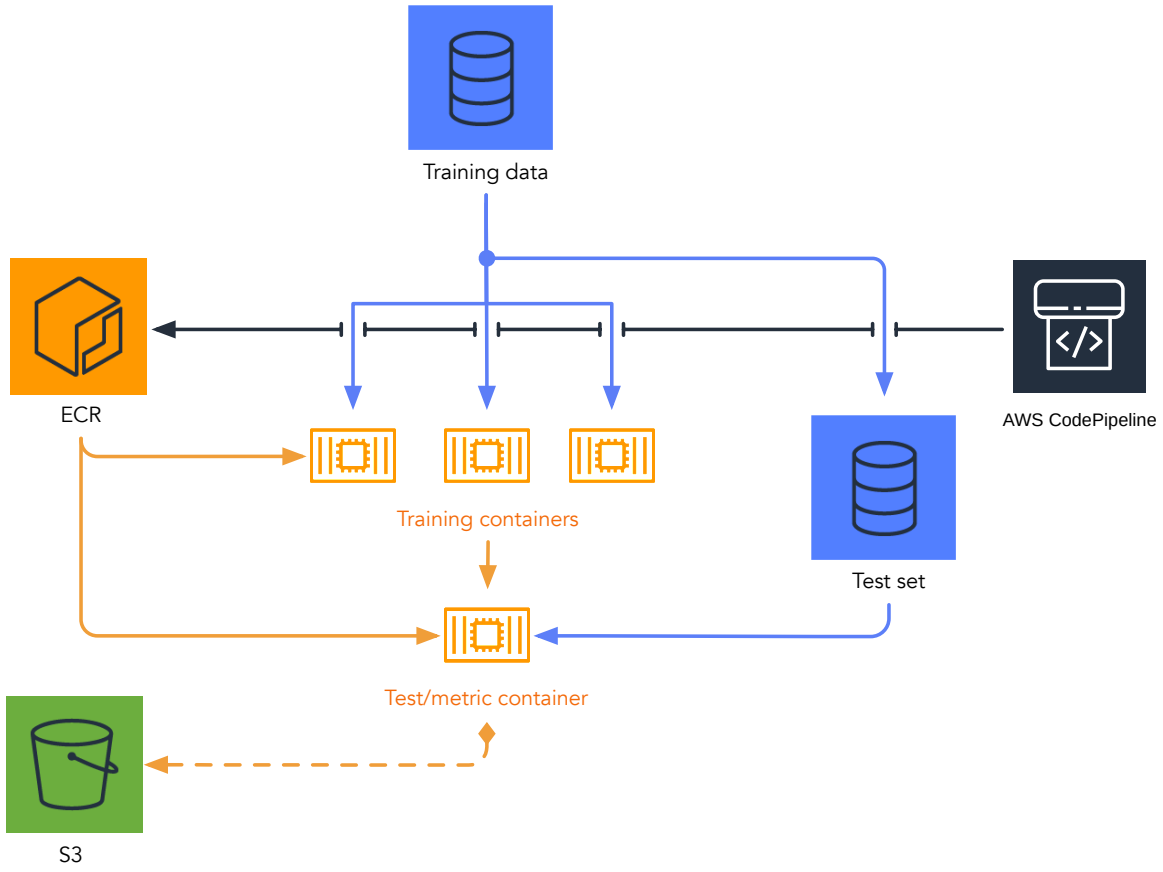

Figure 6. Multi-container training. A number of containers running in parallel each train a model. The test/metric container acts as a discriminator, identifying the best-performing model over the test set.

Alternatively, multi-model and multi-hyperparameter training with comparison can be accomplished within a single container, along with determination of model performance. However, such an arrangement does not have the strict guarantees against data leakage and does not implement the strict separation of concerns between containers that multi-container parallel training does. In addition, parallel training can often deliver results significantly faster, especially if the training process is non-trivial, such as training large and complex neural networks.

Conclusion

The SAIL methodology presented in this paper allows data science and machine learning teams to reap the benefits of tools and methodologies developed initially for the DevOps field. The use of efficient data science and machine learning pipelines guarantees reproducibility and consistency while also automating many of the tedious workflow management tasks. In addition, the SAIL approach to decoupling training and inference pipelines ensures that both of those processes can proceed on their own appropriate schedule.

Implementing the SAIL approach in practice is not difficult, and once the cornerstones of the architecture are implemented, it is relatively self-driving. In our experience, various SAIL implementations have radically accelerated data science and machine learning operations, while requiring relatively little additional resources to establish the initial operational framework. For enterprises of all sizes that are considering ways to operationalize data science, the SAIL architecture offers a solid starting point for designing highly reliable, scalable and adaptable data science and machine learning pipelines that can be deployed on a wide range of platforms, systems and architectures.

About the Authors

As CTO, Tamás Földi leads the Starschema technical team and ensures the best technologies for the jobs are used and the most efficient methods are implemented. Tamas is an active advocate of data, current Zen Master of Tableau for the fourth consecutive year, and considered one of the top data consultants in the world.

Prior to co-founding Starschema to help solve the toughest business data challenges, Tamas worked as a white hat hacker and professional consultant for Fortune 500 clients.

Chris Csefalvay is Starschema's VP for Special Projects, having previously served as Principal Data Scientist at Starschema. As a data scientist with over 10 years' experience, he has pioneered AI approaches in epidemiology, earth observation and digital signal processing. Educated at Oxford and Cardiff, he has worked in data science roles for companies across Europe and the Americas and holds a number of patents in the field of machine learning, AI and DSP.

About Starschema

At Starschema we believe that data has the power to change the world and data-driven organizations are leading the way. We help organizations use data to make better business decisions, build smarter products, and deliver more value for their customers, employees and investors. We dig into our customers toughest business problems, design solutions and build the technology needed to compete and profit in a data-driven world.