# Towards a provenance-enabled, reproducible, and extensible machine learning platform by integrating databases, web services, containers, and code repositories in a loosely coupled manner

**R.M. Singh** [a], **P. Wilson** [b], **L. Gregory** [b], **S. Seaton** [a] and **B. Malone** [b]

[a] *CSIRO Land & Water, Canberra, ACT*
[b] *CSIRO Agriculture & Food, Canberra, ACT*
*Email: ramneek.singh@csiro.au*

**Abstract:** To create a provenance-enabled, extensible machine learning (ML) platform is an exercise that can involve different pieces of software: databases which supply the data to create the model and can also potentially store the result of predictions, web services that allow for ingestion of data into databases and also supply data out of the database to the consumer, code repositories that can be used to track the versioning of models for the sake of provenance and last but not the least the machine learning models themselves. Here we present an approach that we used to create a Soil Spectral inference platform by integrating these software components along with the use of Docker to create a technology agnostic, loosely coupled ML platform. The approach we show here demonstrates how metadata stored in a database can be used not only to drive a workflow but also that a database driven approach allows for extension points where new logic can be plugged-in to expand the capability of the platform.

In our approach, the database plays the central role in the application design. It faithfully reflects the entities involved in problem space. Instead of starting off with a more agile approach of creating, or prototyping, applications to reflect the ML workflow, we invested some time beforehand, understanding the domain to tease out the domain objects, their relationships to one other and the likely workflows that the ML platform will support. We also involved potential future external users in key discussions to understand their workflows and make sure our database is flexible enough to accommodate their concepts. In all this, we used the database schema diagram as a key artifact to consolidate the understanding of the domain, to make explicit the relationships between more prominent entities and to communicate with different team members on the project. Once the schema was in a stable enough state that we started designing the applications that would run off it.

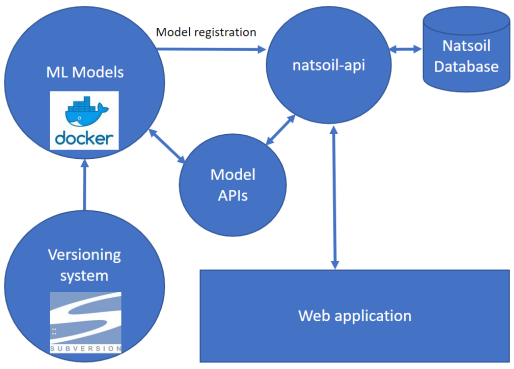


**Figure 1.** Architecture of the ML platform running off a database.

The ML platform applications use the metadata about extensions point stored in the database to implement the machine learning workflow. At the project onset it was decided to use a tech stack that was portable and for those components where we could not control the technologies being used, such as the ML models themselves, we used Docker containers to abstract away the implementation details and expose the model to the platform as a set of interfaces that would remain common for all the models hosted on the platform.

For provenance, we decided to use Subversion, which is an open-source version control system and can handle large files. The ML models and the input data that was used to build the models is versioned in subversion, and that version information is used as ML model metadata in the database to track provenance.

The loosely coupled framework template that we used is modular in nature and the subsystems are connected to each other through web service end points. It can handle future changes by combining, modifying or adding more subsystems visible through the database, each exposing its capabilities through web services.

***Keywords:*** *Machine learning models, databases, web services, model provenance, docker containers*

Singh *et al.*, Towards a reproducible and extensible machine learning platform by integrating databases, web services, containers, and code repositories in a loosely coupled manner

## 1. INTRODUCTION

'Data is the new oil' might come across as a cliché, but there is truth to it if we make use of the data, for example, for drawing inferences, for anomaly detection, or for making predictions. Data sitting in isolation, in a database, by itself is not of much use, and must be made part of an ecosystem to make it valuable. The barriers to the use of data must be lowered and access to it made seamless from applications using it. We must open the database for external access to unlock its value and make up-to-date data easier to use.

CSIRO maintains a Soils database which contains descriptions of over 21,000 soil site investigations from all over Australia. The data includes morphological descriptions, chemical, physical, and mineralogical properties, and spectral predictions, along with soil specimen management data for samples stored in the Australian National Soil Archive facility. Previously when a user wanted access to data, they would place their request with the team managing the database and a custom query would be written to satisfy the data request. This workflow was sub-optimal because the database, in effect, was a silo with no means or mechanisms for convenient programmatic access to it. Satisfying a data request by taking a dump of data meant the end user would have a static snapshot of the data for their request and one that would become stale as new data was added, or existing data modified, in the main database. If the end user was doing modelling with that data as an input, then those models would be running off a stale copy of the data.

Machine learning (ML) models can also be built using the soil data and associated infrared spectra data (namely in the visible-, near- (visNIR) and mid-infrared (MIR) regions of the electromagnetic spectrum), to predict soil properties from soil specimens in a cost-effective manner, and faster than traditional methods of getting soil samples tested at a laboratory. These spectral inference ML models also need to be put into production. The spectral database that will hold the data that drives the models will have linkages to the existing soils database and therefore it is easy to extend the soils database to hold spectral information as well. But that would lead to violation of the design principle of 'separation of concerns'. For example, the soils database was designed to hold soil related information, and if it also starts storing information for hosting a ML platform, then it is pushing its mandate. Moreover, we could also have 2 separate databases, a soils database which maintains soil properties data and a separate spectral database that was used to create the inference models. This does take care of 'separation of concerns' design principle but keeping data in sync across database boundaries brings its own set of problems around data integrity and consistency. The spectral data used to build inference models would need to be tied to soils sample metadata and soil property data in the soils database by the sharing of keys across database boundaries. To keep shared data in two separate databases in sync using referential integrity promised by database systems is not supported as the keys cannot span multiple databases. Hence, for example, one could get into a situation where a particular soil sample record was deleted from the soils database, but it is still being referenced in the spectral database.

To answer the question of provenance and reproducibility, we must track the input dataset that was used to build the ML models, the model-build script (wherever possible) and the model itself. This is another area where there is no best practise answer. The input data used for training the models could be in any format. Not only that, the internal workflow in the model-building stage of a model could be ingesting and processing the data differently from another similar model. For example, some models might expect a combined dataset which includes the predictor and predicted label while some might want to ingest the predictors and predicted label data separately and associate them together in the model build script. What this means is we could not a pre-process the input data into a standard structure to store it into the database as it would affect the reproducibility of the model. We want to have the ability to reproduce the model by feeding in the input data to the model script without making changes to dataflow or logic in the model script. But the downside to not having the input datasets of the models in a standard structure in database is that comparative analysis of the inputs, with respect to different models, is not readily supported out of the box.

For designing a ML system with many sub-components, necessary care must be shown in selection of technologies used for developing the sub-components. Choice of technology stack used could restrict the platforms that such a ML system can run on and hamper future development efforts. Also, the ML models themselves potentially could be developed using a technology that might be tied to a specific platform. For example, a model compiled as a windows executable (.exe) needs Windows Operating system to be able to run. So how do we allow components like these to interplay with each other regardless of choice of their tech stack. We not only need to use technologies that are platform independent but also need to accommodate those components whose choice of technology is incompatible with ours, such as the ML models which are not designed in house.

This paper describes how the team, having a wide range of expertise, came together to design and develop an extension to an existing soils database to create a ML platform that is extensible, technology-stack agnostic for

the ML models, not tied to a particular Operating system for running the platform sub-components, loosely coupled and configurable through a database. The approach we present here should be useful in developing similar systems.

## 2. METHODOLGY

The CSIRO national Soils database is hosted on SQL Server (Ward 2019) in normalized schema. As already stated, there was no convenient way to extract data out of it for modelling purposes without manual intervention. We rectified the situation by writing pipelines to the data so that a modeler could use them fetch the data for their modelling needs. For data egress, we developed JSON web service APIs using which a modeller could fetch latest data from the database for their modelling needs. The advantage to having data at a web service endpoint is that the calls to web service endpoints can be embedded in modelling scripts and thus the control flow remains in a single place, and one does not have to jump 'out of process' for fulfilling the data needs.

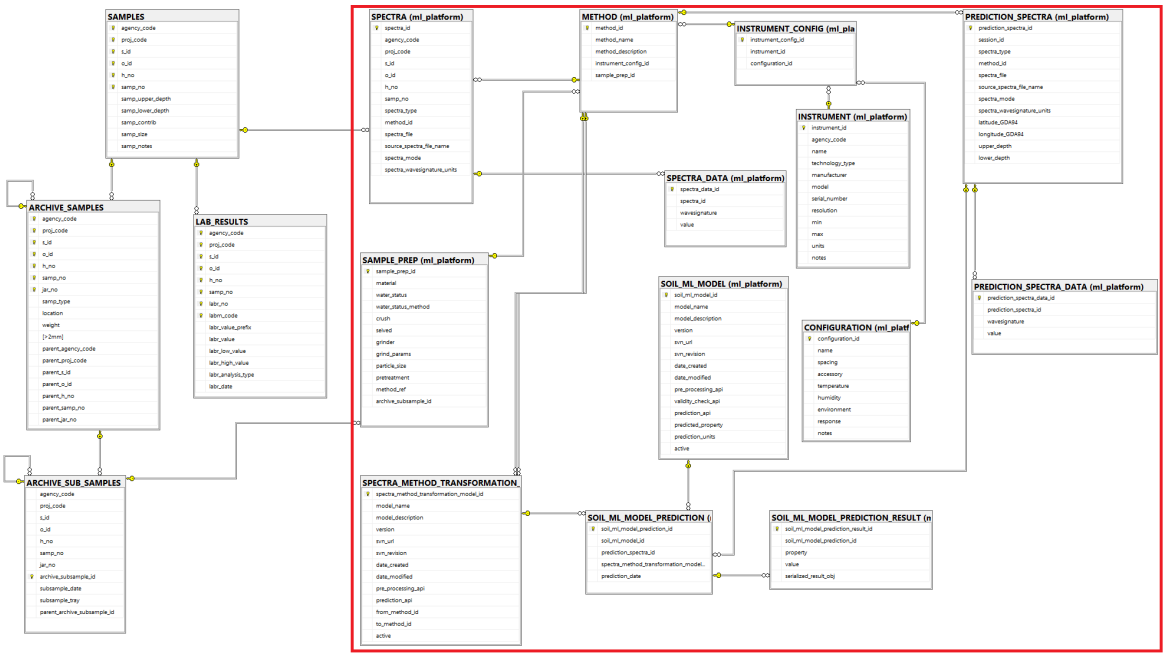


**Figure 2.** Parts of new ML platform database schema, shown in red outline, links with existing database schema (only a part of which is shown) using foreign keys.

To extend the database to enable the development of a machine learning platform, we decided to create a 'modelling' schema. The said new schema has tables for soil spectra as well as ML models for spectra inference of soil properties and holds the prediction results. Having a separate schema for the ML platform is in-line with the design principle of 'separation of concerns'. But since the schema sits in the same existing database and not in a new separate database, we do not have to use techniques like distributed transactions to keep to databases in sync with respect to the shared linkage data. Distributed transactions work fine in some cases but bring their own complexity with them. So, having a new schema is the simplest solution that satisfies our requirements.

For provenance and reproducibility, we decided to store the model creation and related scripts, the input spectra and lab data used to train the model and the serialized model object itself in a subversion repository. Subversion (Pilato et al. 2008) can deal with large files more efficiently than Git (Loeliger 2012) which nowadays is a popular distributed repository. A particular model can be updated or modified numerous times in its lifecycle. For each update of the model, we push the changes again in subversion. As each update will be associated with a revision number tracked by subversion, we can select a particular revision of the model, and its inputs, from subversion for the purpose of provenance.

To deploy the models, we decided to wrap the model functionality in JSON (Marrs 2017) web services with swagger (Surwase 2016) explaining the API specification. And the web service implementation in turn was hosted in a Docker (Merkel 2014) container. This pushes the model behind 2 layers of indirection: first is JSON webservices which makes all the models conform to an interface so that a consumer of the models knows how to interact with the models in a standardized way and the second layer of indirection is added by running the model web services in Docker which allows us to host models in a platform agnostic manner. If the model is a Windows executable, then it has a dependency on Microsoft Windows OS and thus we will host the model in Docker container running Windows and likewise for Linux where needed. Other benefit of running each model in its own separate docker container is isolation. Not only are the models isolated from each other but each of them is isolated from the host operating systems as well. That means if one model fails and corrupts its environment, it won't affect the other models and the system will keep running minus the offending model. Dockerizing the models also provides us with the benefit of portability. We can move individual model containers from one host machine to another, across cloud service providers and move them from virtual machines to physical machines and vice-versa as needed.


**Figure 3.** Swagger API documentation for web services of a particular model.

After the models are deployed using Dockerized web services, we register each model in the database and store metadata about the model such as the model name, model description, the predicted property, units of the predicted property, URL of the subversion repository for the model, the associated subversion revision number, the pre-processing and prediction web service endpoints for the model. This makes the model visible to the database system and application running off the database can 'lookup' information about various models that are available in the system.

A related issue to provenance and reproducibility is provisioning for multiple revisions of the same model. To allow our platform to host multiple revisions of the same model, we added a 'version' metadata field in the database for each model. Although we could have also used the subversion revision number to denote different versions of the model, using a custom number we generate ourselves for the version allows us to keep successive version numbers together without any gaps in between them. This makes versioning updates easier to comprehend for the end user. As prediction results, which are also stored in the database, are associated with a particular version of the model, so from that we can track backwards to a particular revision in subversion and get the associated model object, and scripts, responsible for the result. The system also allows for soft deletion of models from the platform by setting the model as 'inactive'. This method preserves the provenance chain of prediction results which were created in the database as result of execution of that model.

Figure 1 shows the different components in the system. The natsoil-api component, written in ASP.NET core (ASP.NET) to be portable, and also running in Docker, is the data access layer (DAL) providing services for applications built on top of the database. Having one layer, the DAL, to handle data access mechanisms not only allows for easier placement of access controls but also has the advantage of centralizing the data access logic and business logic in one place. That means the ML platform can not only be consumed from a front-end user-interface, but api-level access can also be granted to end users (including external users) who want to interact with the platform in a programmatic manner. End users can even develop their own custom user-interface on top of natsoil-api (Brooks 2016).

Storing the ML Model and as well as the code and input data to build the model in Subversion and having all the components of the platform containerized in Docker gives us reproducibility at various levels of detail in the platform. The model results are reproducible because we store the model that generated the results in subversion and link it in the database, the model itself is reproducible itself as we store the scripts (or source code) as well as the input data used to train the model, and the computing environment of web-services running on the platform is reproducible as they are running in Docker which uses Dockerfile to document as well as build the environment.

```
FROM rstudio/plumber

# RUN R -e "install.packages('broom')"
RUN R -e "install.packages('pls')"
RUN R -e "install.packages('wavethresh')"
RUN R -e "install.packages('Cubist')"
RUN R -e "install.packages('plyr')"
RUN R -e "install.packages('signal')"
RUN R -e "install.packages('httr')"
RUN R -e "install.packages('plyr')"
RUN R -e "install.packages('reshape')"
RUN R -e "install.packages('reshape2')"
RUN R -e "install.packages('stringr')"
RUN R -e "install.packages('MASS')"

RUN apt-get update
RUN apt-get install -y gdal-bin libgdal-dev libudunits2-dev

RUN R -e "install.packages('sp')"
RUN R -e "install.packages('spatialEco')"

COPY ./ /usr/work/

USER root
RUN chmod +x /usr/work/model_api_script/pHc_model_plumber.R
RUN chmod +x /usr/work/model_api_script/pHc_model.R
RUN chmod +x /usr/work/model_scripts/spectralProcess_functions.R
RUN chmod +x /usr/work/model_scripts/bagging_functions_plsr.R

#CMD ["ls"]
CMD /bin/bash Rscript /usr/work/model_api_script/pHc_model_plumber.R

EXPOSE 8000
```

**Figure 4.** Example Dockerfile documenting the hosting of a model web service.

JSON is glue which binds different web services in the platform. The data access layer (DAL), front-end applications, prediction models and ingestion/consumption workflows, all make use of JSON web services where possible to interact with each other. Internally the DAL JSON web services make use of T-SQL stored procedures (Ben-Gan et al. 2015) for data access. Stored procedures help reduce a lot of network traffic as only the procedure name and input parameters are passed over the network. The procedure logic is executed in the database and only the final result is sent back to the caller. Had we, instead, implemented the procedure logic in DAL layer, it would have caused more round trips between the DAL layer and database resulting in more network traffic.

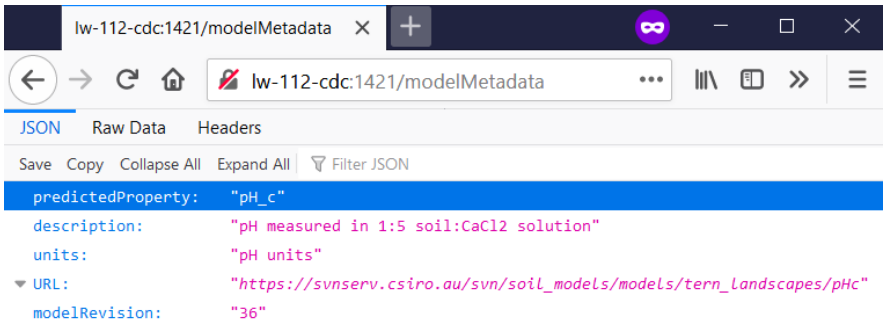


**Figure 5.** A model web-service endpoint returning metadata about itself in JSON.

Deploying the logic in stored procedures also helps in encapsulating the logic and, thus, making it modular and allows you to hide the complexity. They also promote efficient plan caching. The alternative to stored procedures, other than ad-hoc stand along queries, would have been to use an Object-relational mapping (ORM) layer. The advantage of using an ORM layer is that it bridges the impedance mismatch between the objects in a programming environment like ASP.NET and database tables and make it easier to develop CRUD style applications. But on the other hand, they prevent the centralization of business logic by having the logic embedded in the application layer built on top of database. And that is not a good pattern if maintainability of the solution is brought into question. Applications sitting on top of a database can be written and re-written and if the business logic also sits in the database layer, it makes the job of application upgrades and rewrites much easier. Also, it is easier to tune queries written by hand than those produced by ORM.

Figure 6 below shows the prediction request for serviced by a ML model hosted on the platform. End user of the platform will upload the spectra file consisting of wavelength/value pairs to the prediction endpoint for a model by making a POST request. The return result of the POST request would be the prediction result object formatted as JSON. In this case, the model has predicted that for the given input spectra, the pH value is 4.4.

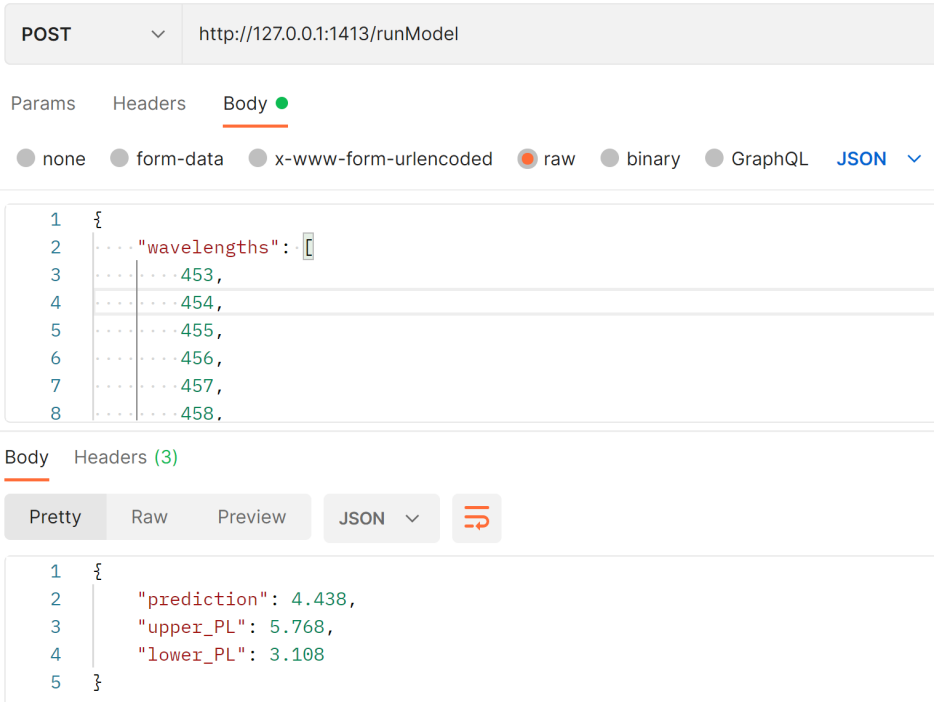


**Figure 6.** *Runmodel* web-service endpoint of a model returning the predicted result.

## 3. FUTURE WORK

Following the philosophy of avoiding premature performance optimization, we have not used tools and methodologies that can help scale up the application to handle influx of prediction requests. But the way we have designed the system, it would straightforward to scale it up. At a basic level, we have designed the prediction services to run inside docker containers using docker compose. If at some point in future we start hitting performance bottlenecks due to a large number of prediction requests, we can explore scaling and load balancing using docker compose or we can also evaluate the use of Kubernetes. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. The modular nature of ML platform with the components being connected to each other through json web services allows us the option to scale horizontally as well as vertically as the individual components can be moved around in a flexible manner.

## 4. DISCUSSION AND CONCLUSION

This paper described some of the issues involved around provisioning a ML platform on an existing database and the method we used to design one which addresses the key question of provenance and reproducibility. The project team had members with a varied background and skills. By working in an agile manner, it was possible to formulate a preliminary design and then develop a system based on that design. Along the way, we adjusted the system to accommodate for change in requirements as our understanding of the domain increased. Having prospective external customers also involved in the initial development process provided us with much needed feedback and helped us align the system with user expectations. We hope the process we have described here would serve as a blueprint for design of other similar ML platforms.

## ACKNOWLEDGMENTS

This work was funded by the Soil Spectral Platform Digiscape project. The authors acknowledge contributions from all members of the Soil Spectral Platform Digiscape project team.

## REFERENCES

Ward B. (2019). SQL Server 2019 Revealed. Apress, Berkeley, CA.

Pilato, C. M., Collins-Sussman, B., & Fitzpatrick, B. W. (2008). Version control with subversion: next generation open source version control. " O'Reilly Media, Inc.".

Loeliger, J., & McCullough, M. (2012). Version Control with Git: Powerful tools and techniques for collaborative software development. " O'Reilly Media, Inc.".

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux journal, 2014(239), 2.

Marrs, T. (2017). JSON at work: practical data integration for the web. " O'Reilly Media, Inc.".

Surwase, V. (2016). REST API modeling languages-a developer's perspective. Int. J. Sci. Technol. Eng, 2(10), 634-637.

Brooks, G. (2013). Benefits of APIs. Digital.Gov. https://digital.gov/2013/03/12/benefits-of-apis/

ASP.NET. (n.d.). Microsoft Docs. https://docs.microsoft.com/en-us/aspnet/core/

Ben-Gan, I., Machanic, A., Sarka, D., & Farlee, K. (2015). T-SQL Querying. Microsoft Press.