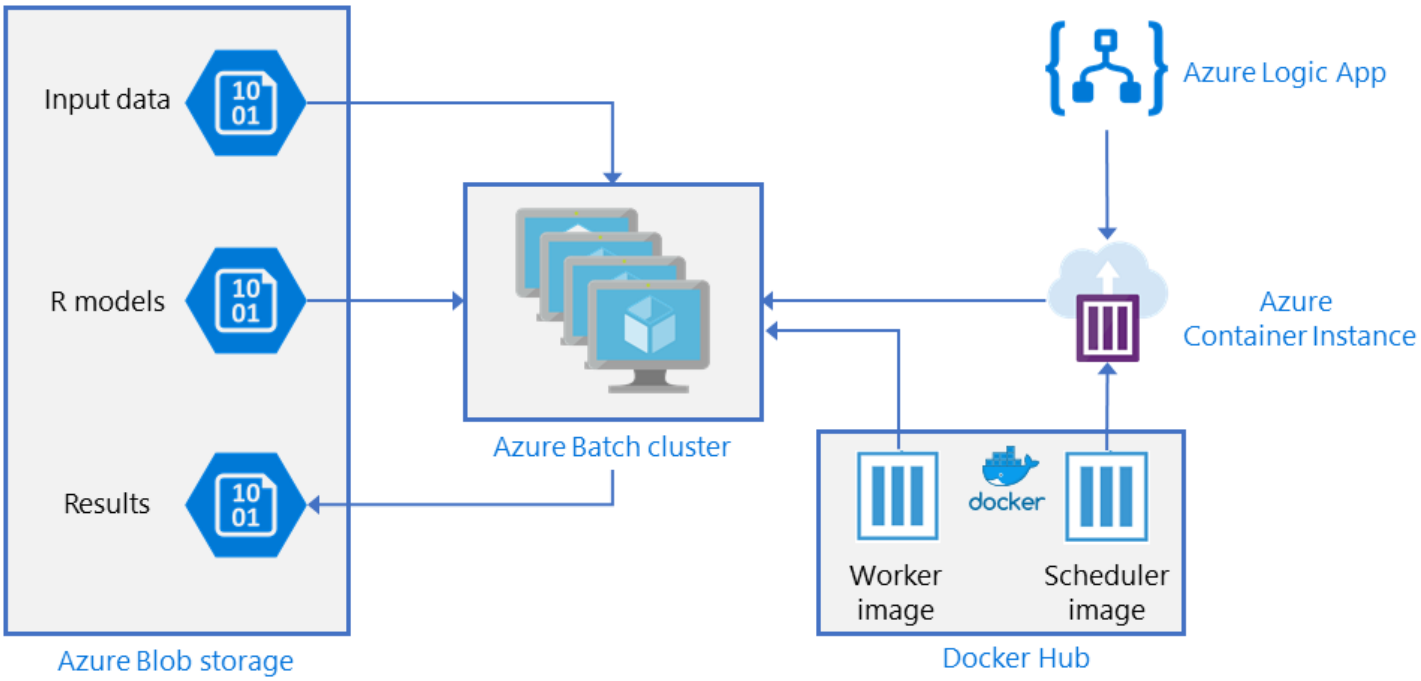
Batch scoring of R machine learning models on Azure

03/29/2019 • 5 minutes to read • Contributors

In this article

This reference architecture shows how to perform batch scoring with R models using Azure Batch. The scenario is based on retail store sales forecasting, but this architecture can be generalized for any scenario requiring the generation of predictions on a large scaler using R models. A reference implementation for this architecture is available on GitHub.



Scenario: A supermarket chain needs to forecast sales of products over the upcoming quarter. The forecast allows the company to manage its supply chain better and ensure it can meet demand for products at each of its stores. The company updates its forecasts every week as new sales data from the previous week becomes available and the product marketing strategy for next quarter is set. Quantile forecasts are generated to estimate the uncertainty of the individual sales forecasts.

Processing involves the following steps:

1. An Azure Logic App triggers the forecast generation process once per week.
2. The logic app starts an Azure Container Instance running the scheduler Docker container, which triggers the scoring jobs on the Batch cluster.
3. Scoring jobs run in parallel across the nodes of the Batch cluster. Each node:

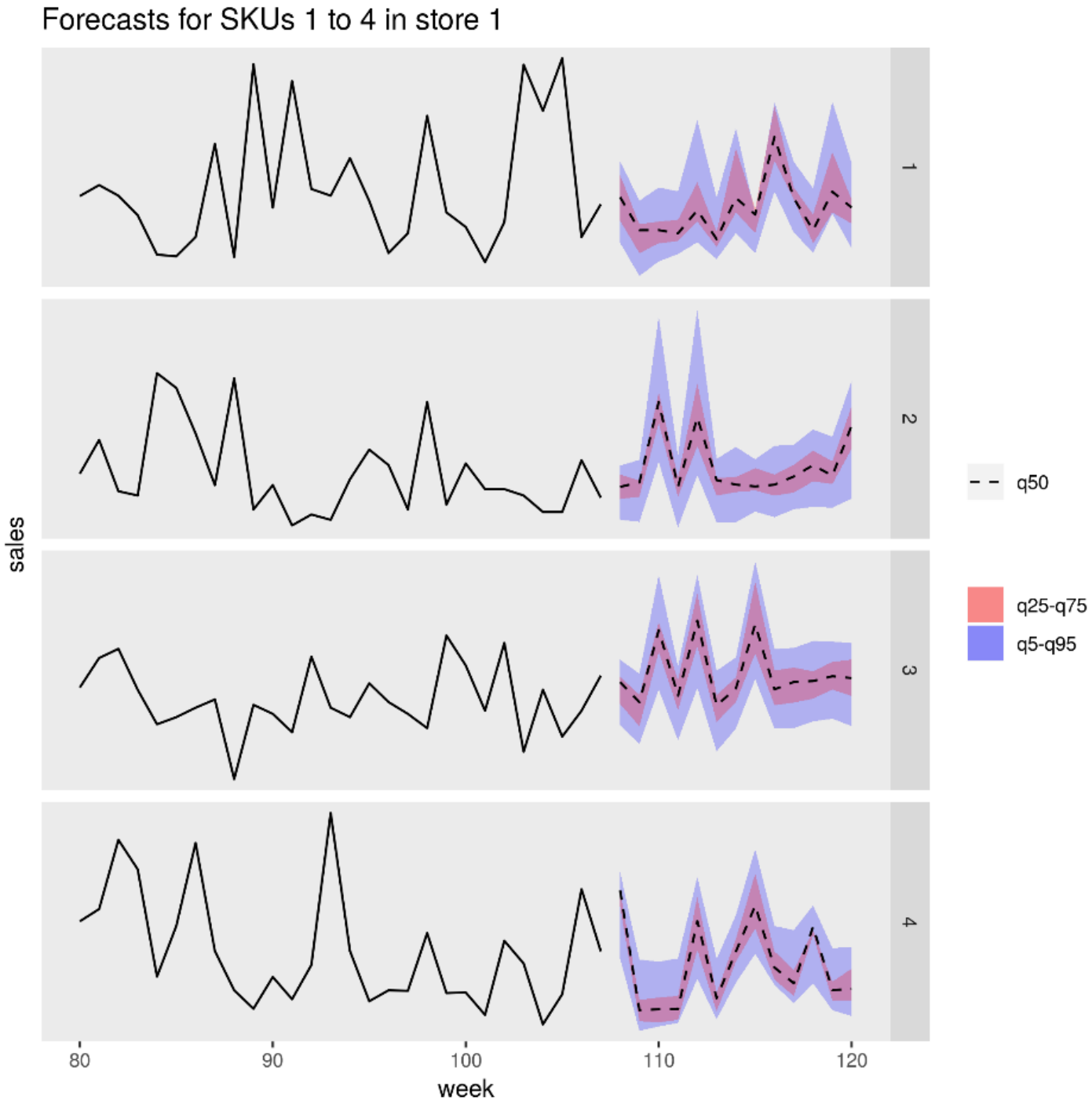
    a. Pulls the worker Docker image from Docker Hub and starts a container.

b. Reads input data and pre-trained R models from Azure Blob storage.

c. Scores the data to produce the forecasts.

d. Writes the forecast results to blob storage.

The figure below shows the forecasted sales for four products (SKUs) in one store. The black line is the sales history, the dashed line is the median (q50) forecast, the pink band represents the twenty-fifth and seventy-fifth percentiles, and the blue band represents the fifth and ninety-fifth percentiles.



Architecture

This architecture consists of the following components.

Azure Batch is used to run forecast generation jobs in parallel on a cluster of virtual machines. Predictions are made using pre-trained machine learning models implemented in R. Azure Batch can automatically scale the number of VMs based on the number of jobs submitted to the cluster. On each node, an R script runs within a Docker container to score data and generate forecasts.

Azure Blob Storage is used to store the input data, the pre-trained machine learning models, and the forecast results. It delivers very cost-effective storage for the performance that this workload requires.

Azure Container Instances provide serverless compute on demand. In this case, a container instance is deployed on a schedule to trigger the Batch jobs that generate the forecasts. The Batch jobs are triggered from an R script using the doAzureParallel package. The container instance automatically shuts down once the jobs have finished.

Azure Logic Apps trigger the entire workflow by deploying the container instances on a schedule. An Azure Container Instances connector in Logic Apps allows an instance to be deployed upon a range of trigger events.

Performance considerations

Containerized deployment

With this architecture, all R scripts run within Docker containers. This ensures that the scripts run in a consistent environment, with the same R version and packages versions, every time. Separate Docker images are used for the scheduler and worker containers, because each has a different set of R package dependencies.

Azure Container Instances provides a serverless environment to run the scheduler container. The scheduler container runs an R script that triggers the individual scoring jobs running on an Azure Batch cluster.

Each node of the Batch cluster runs the worker container, which executes the scoring script.

Parallelizing the workload

When batch scoring data with R models, consider how to parallelize the workload. The input data must be partitioned somehow so that the scoring operation can be distributed across the cluster nodes. Try different approaches to discover the best choice for distributing your workload. On a case-by-case basis, consider the following:

- How much data can be loaded and processed in the memory of a single node.
- The overhead of starting each batch job.
- The overhead of loading the R models.

In the scenario used for this example, the model objects are large, and it takes only a few seconds to generate a forecast for individual products. For this reason, you can group the products and execute a single Batch job per node. A loop within each job generates forecasts for the products sequentially. This method turns out to be the most efficient way to parallelize this particular workload. It avoids the overhead of starting many smaller Batch jobs and repeatedly loading the R models.

An alternative approach is to trigger one Batch job per product. Azure Batch automatically forms a queue of jobs and submits them to be executed on the cluster as nodes become available. Use automatic scaling to adjust the number of nodes in the cluster depending on the number of jobs. This approach makes more sense if it takes a relatively long time to complete each scoring operation, justifying the overhead of starting the jobs and reloading the model objects. This approach is also simpler to implement and gives you the flexibility to use automatic scaling—an important consideration if the size of the total workload is not known in advance.

Monitoring and logging considerations

Monitoring Azure Batch jobs

Monitor and terminate Batch jobs from the **Jobs** pane of the Batch account in the Azure portal. Monitor the batch cluster, including the state of individual nodes, from the **Pools** pane.

Logging with doAzureParallel

The doAzureParallel package automatically collects logs of all stdout/stderr for every job submitted on Azure Batch. These can be found in the storage account created at setup. To view them, use a storage navigation tool such as Azure Storage Explorer or Azure portal.

To quickly debug Batch jobs during development, print logs in your local R session using the getJobFiles function of doAzureParallel.

Cost considerations

The compute resources used in this reference architecture are the most costly components. For this scenario, a cluster of fixed size is created whenever the job is triggered and then shut down after the job has completed. Cost is incurred only while the cluster nodes are starting, running, or shutting down. This approach is suitable for a scenario where the compute resources required to generate the forecasts remain relatively constant from job to job.

In scenarios where the amount of compute required to complete the job is not known in advance, it may be more suitable to use automatic scaling. With this approach, the size of the cluster is scaled up or down depending on the size of the job. Azure Batch supports a range of auto-scale formulae which you can set when defining the cluster using the doAzureParallel API.

For some scenarios, the time between jobs may be too short to shut down and start up the cluster. In these cases, keep the cluster running between jobs if appropriate.

Azure Batch and doAzureParallel support the use of low-priority VMs. These VMs come with a significant discount but risk being appropriated by other higher priority workloads. The use of these VMs are therefore not recommended for critical production workloads. However, they are very useful for experimental or development workloads.

Deployment

To deploy this reference architecture, follow the steps described in the GitHub repo.