Cloud computing for voxel-wise SEM analysis of MRI data

Joshua N. Pritikin

Virginia Commonwealth University

J. Eric Schmitt

Hospital of the University of Pennsylvania

Michael C. Neale

Virginia Commonwealth University

Author Note

Joshua N. Pritikin, Department of Psychiatry and Virginia Institute for Psychiatric and Behavior Genetics, Virginia Commonwealth University; J. Eric Schmitt, Departments of Radiology and Psychiatry, Hospital of the University of Pennsylvania; Michael C. Neale, Departments of Psychiatry and Human & Molecular Genetics, Virginia Institute for Psychiatric and Behavior Genetics, Virginia Commonwealth University.

This research was supported in part by National Institute of Health grants R01-DA018673 and R25-DA026119 (PI Neale). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Institutes of Health.

Correspondence concerning this article should be addressed to Joshua N. Pritikin, Department of Psychiatry and Virginia Institute for Psychiatric and Behavior Genetics, Virginia Commonwealth University, 800 E. Leigh St., Richmond, VA 23219. E-mail: jpritikin@pobox.com

Abstract

As data collection costs fall and vast quantities of data are collected, data analysis time can become a bottleneck. For massively parallel analyses, cloud computing offers the short-term rental of ample processing power. Recent software innovations have reduced the effort needed to take advantage of cloud computing. To demonstrate, we replicate a voxel-wise examination of the genetic contributions to cortical development by age using evidence from 1,748 MRI scans. Specifically, we employ off-the-shelf Kubernetes software that permits us to re-run our analyses using almost the same computer code as was published in the original article. Large, well funded institutions may continue to maintain their own computing clusters. However, the modest cost of renting and ease of utilizing cloud computing services makes unprecedented compute power available to all researchers, whether or not affiliated with a research institution. We expect this to spur innovation in the sophisticated modeling of large datasets.

Keywords: cloud computing; neuroimaging; genomic SEM; big data; GWAS; fMRI; ecological momentary assessment; time series; developmental; genetics; twins; methylomics

Cloud computing for voxel-wise SEM analysis of MRI data

Introduction

Ever growing volumes of data entail more and more compute time to analyze. Spatial and temporal resolutions of neuroimaging continue to increase (e.g., Uğurbil et al., 2013), the cost of whole genome sequencing continues to plummet (e.g., Plöthner, Frank, & von der Schulenburg, 2017), and SNP chips permit inexpensive assessment of common genetic polymorphisms (e.g., Ha, Freytag, & Bickeboeller, 2014). Data volumes are further multiplied when data are collected longitudinally, and especially when measures are sampled repeatedly with technologies such as electroencephalography or functional Magnetic Resonance Imaging (fMRI). With so much data, an uncomfortable amount of time can be required to fit models. Researchers may choose to rely on institutional computer clusters, but these clusters are expensive to operate and require ongoing maintenance and upgrades, regardless of whether they are being fully utilized or not. Cloud computing – the computational equivalent of a rental apartment – is expensive, but you only pay for what you use. The present article examines how to execute massively parallel structural equation modeling jobs in the cloud.

There are a variety of tasks involved in the execution of large parallel jobs: hardware is provisioned, a list of work is generated, work is assigned to nodes, nodes report results or errors back, and errors can be retried. All this needs to happen reliably. When an error occurs, the precise circumstance and root cause must be accurately reported. Software to manage large parallel workflows is hard to test because a realistic test consumes a lot of resources. Prior work has relied on specialized software to manage parallel computation (e.g., Kenny et al., 2009). What is novel about the current approach is how little software development is required. All components are prebuilt, thoroughly tested, and merely fit together skillfully. High performance computing is one situation where the wisdom of *less is more* applies – less custom software enables more productivity and reliability.

To demonstrate our approach, we replicate a study that examined changes in the

genetic contributions to cortical development by age (Schmitt et al., 2014). These data consist of 81,924 neuroimaging vertices acquired from 792 normally developing children, adolescents, and young adults from 410 families using a General Electric 1.5 Tesla Signa Scanner. Up to eight MRI scans were performed per individual. The mean interval between scans was 2.4 yrs. A total of 1,748 MRI datasets were acquired. As part of the online supplement, the data used in this study are available from the Dryad Digital Repository (http://datadryad.org).

Model

A traditional univariate twin model (Figure 1) uses known differences in zygosity to decompose phenotypic variance into additive genetic (A), shared environmental (C), and unique environmental (E) variance based on differences in observed covariance between different types of family relationships (Neale & Cardon, 1992). Often the variances of A, C, and E latent variables are fixed to 1 and the variance decomposition is implied by the path coefficients. Long and painful experience with numerical optimizers have taught researchers to minimize correlation between parameters, to eliminate boundaries, and to try to keep parameters on similar scales (Luenberger & Ye, 2008). Model parameterization details are often motivated by these numerical considerations. Figure 1 was extended in two ways. First, outcome Y need not be observed. The model works equally well if Y is latent. Second, essentially the same model was extended to more than one outcome, decomposing a covariance matrix instead of a single variance. Conveniently, a Cholesky factor can be estimated directly using path coefficients. The use of a Cholesky factor eliminates the use of technically challenging methods such as non-linear constraints to preserve covariance positive definiteness.

A non-genetic longitudinal growth curve model (Figure 2) uses repeated measures to estimate changes in means and variances with time (Duncan & Duncan, 2004). Path coefficients from the latent factors $\sigma^2_{intercept}$ and σ^2_{linear} to the observed variables x are fixed.

All the paths from $\sigma^2_{intercept}$ are fixed to 1.0 and the paths from σ^2_{linear} follow a linear trend. Figure 2 was modified in three ways. First, to increase the plausibility of the model, we allowed the possibility of a quadratic variance component. Second, instead of a unit time increment between observations, we set the path coefficients to reflect the actual time between observations (in days) for each participant. The path coefficients were set to time and time squared for σ^2_{linear} and $\sigma^2_{quadratic}$, respectively. Finally, instead of estimating the measurement error as a variance, we fixed the variance to 1.0 and estimated the path coefficient. This parameterization obviated bounds on the variance parameter to ease optimization.

These two models (Figures 1 and 2) were combined into a genetically informative quadratic latent growth curve model (Figure 3). Rectangles represent the observed measures of cortical thickness of a given vertex k for time point i from subject n. Circles represent additive genetic (A), unique environmental (E), and measurement error (ϵ) latent factors, all with variance fixed to unity. In view of the results of prior studies, common genetic variance (C) was omitted from the model. Double headed (dotted) arrows depict correlations between latent factors, with $\alpha = 1$ for additive genetic MZ correlations and $\alpha = 0.5$ for DZ twin and sibling genetic correlations. All single-headed paths from these factors are freely estimated parameters. Ovals represent the latent growth component of the model. For simplicity, Figure 3 only depicts path coefficients from 2 subjects in a family; in actuality the model included up to $n = 5$ relatives and $i = 8$ occasions, for a maximum 40×40 covariance matrix of observed measures for each k. This model was specified in OpenMx (Neale et al., 2016), an R package for structural equation and other statistical modeling.

From the foregoing description, it should be clear that, although the model is a complex structural equation model, there is nothing really extraordinary about it. Every detail of the model can be understood and specified just like any other structural equation model. Hence, without loss of generality, this model was instantiated and fit, by maximum

likelihood, to the data on each contralateral voxel. Several submodels testing for the significance of genetically and environmentally-mediated effects on growth were also fitted. On current Intel x86_64 hardware, it takes about 3 minutes for one CPU core to process one voxel. By leveraging cloud computing to rent 1024 CPUs, these 40,924 sets of analyses were completed in approximately two hours, at a cost of about $200 USD.

Method

Overview

Cloud computing is an information technology paradigm that provides the illusion that physical hardware is entirely managed by software. Physical hardware can be rented, configured, used, and discarded as directed by a web form or scripts as easily as any other computer task, such as sorting a list of names alphabetically. Kubernetes provides an open-source cloud service abstraction layer. Many cloud providers offer a Kubernetes interface including Google Cloud Services; IBM BlueMix; RedHat OpenShift; Microsoft Azure; and Amazon Web Services. An up-to-date list of providers is available at https://kubernetes.io/docs/setup/. Each provider offers different degrees of customization. Since our needs are simple (compared to, e.g., running a large website like https://fsf.org/), we use a so-called *hosted solution* that involves a minimum of configuration.

The approach described here closely follows Cook (n.d.). We use the components `foreach` (Analytics & Weston, 2015), `doRedis` (Lewis, 2015), `Redis`, Docker, and Kubernetes. Figure 4 exhibits the relationships between these components. Here we demonstrate the use of both Google Cloud Services and IBM BlueMix.

Technical Jargon. Kubernetes employs elaborate specialized terminology. In brief, *containers* are the finest indivisible unit. A container runs a single Docker image and can be regarded as a virtual machine. A docker image consists of a file system and some meta information. Kubernetes organizes a set of related containers into what is called a *pod*. Pods are assigned to *nodes*. Nodes represent physical hardware. A *cluster* consists of a set

of nodes. Clusters are created using a vendor-specific interface. Kubernetes supervises what goes on within a cluster.

Changes for Docker and Kubernetes

The online supplement for Schmitt et al. (2014) contains 639 lines of `R` code to conduct the data analyses. This code was mostly preserved unchanged. About 20 lines were deleted and 48 lines added to the file `template.R`. Here we detail the purpose of these changes. The changes were made in small steps to support careful testing and ensure that no bugs were introduced.

```
1  recordOutput <- function(ign, result) {
2      write.table(result,file="LGCout.txt",col.names=F,row.names=F,
3      na=".",sep="␣",append=TRUE)
4  }
5  registerDoRedis(host='redis', queue='jobs')
6  result <- foreach(i=voxseg,
7                    .packages=c('plyr','reshape','OpenMx'),
8                    .init="ign", .combine=recordOutput, .inorder=FALSE,
9                    .verbose=FALSE) %dopar% processOneVoxel(i)
10 removeQueue('jobs')
```

Lines 1-4 define the `recordOutput` function. This function writes a line of summary output, the contents of the R object `result`. Line 5 creates a database table or spreadsheet to track job status in the `Redis` database. The variable `voxseg` contains a list of voxel numbers, and the `processOneVoxel` function (line 9) fits the model to the data of a given voxel. The `for` loop over each voxel in the original code is replaced by a `foreach` loop; the `foreach` and `doRedis` packages (lines 6-9) enable parallel processing of the loop. The `foreach` function can operate in two modes, namely `%do%` and `%dopar%` (line 9). While both offer almost the same interface, the `%do%` mode executes tasks serially while `%dopar%` executes in parallel. The `%do%` mode is useful for debugging, because errors occur within the same process and do not need to be sent across a network. Once the `R` code works with

serial execution, it can be converted to parallel mode by replacement of `%do%` with `%dopar%`. Computation is then distributed across different processors, but the results are collated. The way this works is that i) the `foreach` function receives arbitrary `R` data back from its workers, and ii) it calls the `.combine` function (line 8). Hence, output can be recorded in a central place (line 1). Once the loop has processed all of the voxels, (line 10) removes the queue.

The `doRedis` approach to parallel `foreach` needs to have the `Redis` database server running to communicate the status of parallel execution. Before attempting to execute tasks in the cloud, there is an opportunity to further debug on a local system. On most Debian-family distributions, the `Redis` database can be installed with the command `sudo apt install redis`. By default, `Redis` creates some empty databases and allows connections from the local host. This installation permits the cloud user to pre-test using their own local system.

For parallel execution on the local host, we call `doRedis::startLocalWorkers(n=2)`. This function forks separate worker processes to present us with exactly the same execution environment that we will face in the cloud. Debugging can be tricky because the usual debugging tools, like `browser()`, are not well suited to the examination of interactions between processes and `doRedis` is not thoroughly meticulous about reporting error conditions. However, there were only a few bugs remaining due to our careful step-by-step approach to changes. We do not give the details of the bugs here because they were idiosyncratic to this particular use and author. Once things were tested and found to operate locally, we interrupt execution with Control-C and turn to the task of making a Docker image for cloud computing.

```
11 FROM openmx/openmx:master
12 RUN apt-get update && apt-get install -y redis-tools
13 RUN mkdir /rsetup
14 ADD ./rsetup.R /rsetup/rsetup.R
15 RUN Rscript /rsetup/rsetup.R
```

```
16 ADD ./DynamicHeritability2.RData /root/DynamicHeritability2.RData
17 ADD ./Rprofile_worker /root/.Rprofile
18 ADD concatdata.R submodels.R voxAC.R /root/
19 ENV OMP_NUM_THREADS 1
20 CMD R --no-save
```

A `Dockerfile` (lines 11-20) is used to specify a Docker image. The command `docker build` takes the `Dockerfile` as input and creates the corresponding image artifact. The base image contains everything needed to run `OpenMx` (line 11). This base image, which can be inspected at https://hub.docker.com/r/openmx/openmx/, is built on top of `r-base`, which in turn is built on top of `debian:testing`. Base images are often maintained by different sets of people. This deep layering is a typical organization strategy of Dockerfiles; it allows us to leverage the work of many talented developers.

Line 12 installs the `redis-tools` package, in the event that we need to troubleshoot something related to `Redis`. Some needed `R` packages are installed (lines 13-15). Raw data downloaded from the Dryad Repository consisted of a single 601 MiB `R` image file named `DynamicHeritability2.RData`. When the analysis scripts are run in the cloud, each compute node needs its own copy of the data. One convenient way to arrange for this is to include the data directly in the Docker image (line 16). For analyses where the data are too large to include in the Docker image, care should be taken to minimize the overhead incurred by node access to the data. Typically overhead is minimized by physically locating the storage close to the Kubernetes cluster. There are both vendor-specific options such as AWSElasticBlockStore (Amazon), GCEPersistentDisk (Google), and AzureDisk (Microsoft); and generic, do-it-yourself options such as NFS, CephFS, and Glusterfs. Kubernetes support for persistent volumes is an evolving area of the specification. New projects will benefit from a fresh look at best practices.

By default, `R` automatically loads and executes `/root/.Rprofile` when it starts. Here, our custom `.Rprofile` (line 17) calls the function `redisWorker`, which connects to `Redis` and which will process work items assigned by `foreach` (see Appendix A). Line 18

places the unmodified analysis scripts from the Schmitt et al. (2014) online supplement into a known location. Next, in line 19, the environment variable OMP_NUM_THREADS is set to limit processing to a single thread, i.e., one part of a single compute core that has multithreading enabled. Parallel processing is managed by `doRedis` so all workers are single threaded.[1] Finally, the Docker image will start `R` immediately upon launch (line 20). The command `docker build` prepares an image, and the command `docker tag` assigns an arbitrary name, `jpritikin/worker`, to it. Lastly, the image is copied to https://hub.docker.com using `docker push`.

On a per-work item basis, `doRedis` provides an empty environment devoid of data. To get our data into memory, we could load it from disk. However, since our data is small enough to fit into RAM, a more efficient solution is to load it once at start up time. This can be accomplished by a small addition to `template.R`,

```
21 if (!("dt_l" %in% ls(envir=globalenv()))) {
22   load("/root/DynamicHeritability2.RData"), envir=globalenv())
23   rm(dt_r, envir=globalenv())
24 }
25  lapply(c('dt_l','demo'), function(n)
26       assign(n, get(n, envir=globalenv())))
```

In line 21, we check whether the data are already loaded. If not, we load them (line 22), and discard the contralateral data that are omitted from the present analyses (line 23). These data are copied into the current environment from the global environment in line 25. No further changes are required to `template.R`. Note that the Docker image does not contain `template.R`. This file will be copied to the cloud by hand, as detailed below in the Execution section.

[1] For advanced uses, multithreaded workers are possible up to the maximum number of CPUs per node.

Execution

We now turn to the practical execution of massive parallel SEM. A narrated video demonstration of this process is available, https://www.youtube.com/watch?v=sDlmjHL_ib8.

Prepare Cluster. To obtain an account on a cloud service provider, we either login to an existing account or go through the registration process and provide payment information. Once authorized, we create a Kubernetes cluster (Figures 5 and 6). There is no need to create more than two nodes at the outset because the cluster can be resized dynamically. Some vendor-specific configuration is needed to tell the local Kubernetes client, *kubectl*, where to find the newly created cluster. Once kubectl is configured, we can launch our pods. It is convenient to specify pod configuration in *yaml*-formatted files Regardless of the specific analyses, we use the same `Redis` pod specification (see Appendix B). The configuration for our `rworker.yaml` pod is as follows,

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: rworker-rc
  labels:
    name: rdocker-hpc-worker
spec:
  replicas: 1
  selector:
    name: rworker
  template:
    metadata:
      labels:
        name: rworker
    spec:
      containers:
      - name: rdockerhpc-worker
```

```
44          image: jpritikin/worker
45          resources:
46            requests:
47              cpu: "600m"
48              memory: "1.4Gi"
```

We provide the name of our Docker image (line 44). A request of 600 milli-CPU (line 47) and 1.4Gi of RAM (line 48) will persuade Kubernetes to provision about 1 CPU per rworker. Depending on the exact configuration of your nodes, some quantity of CPU and RAM resources will be available. You can control how many rworker instances will fit onto your nodes by fine-tuning the resource demands in the `rworker.yaml` file. The vendor-specific status console will likely assist by issuing a warning when more resources are requested than are available. Line 40 names the pods rworker-xxxxx where xxxxx is some random gibberish. At this point, we create the pods using kubectl from the command line:

```
49 kubectl create -f redis.yaml
50 kubectl create -f redis_service.yaml
51 kubectl create -f rworker.yaml
```

This results in the following cluster state, which can be seen with `kubectl get all`:

```
52 NAME                    READY     STATUS    RESTARTS   AGE
53 po/redis-svc-ggcnw      1/1       Running   0          5m
54 po/rworker-rc-g196d     1/1       Running   0          1m
55
56 NAME             DESIRED   CURRENT   READY     AGE
57 rc/redis-svc     1         1         1         5m
58 rc/rworker-rc    1         1         1         1m
```

Commence Analyses. Our pods are running, but we still need to kick off the computation. This is accomplished with a few manual steps. First, we set a shell variable to the name of the first worker, `wk=rworker-rc-g196d`.[2] We will subsequently write `$wk`

[2] This is Bourne shell syntax. Csh and other shells may use a different syntax.

when we want to name this worker. In addition to being a `foreach` worker, this first node will also distribute work and collect results (i.e., run the `foreach` loop).

```
59 kubectl cp template.R $wk:/root/
60 kubectl exec $wk -it bash
61 nohup R --no-init-file --no-save -f template.R &
62 kubectl logs -f $wk
```

The `template.R` script is copied over to the cluster in line 59. Line 60 then opens an ssh session to the node, and line 61 starts the analyses. The nohup command tells the system not to abort running the job if interactive connection is dropped. In line 61, the `-no-init-file` option to R is used to skip the `/root/.Rprofile` initialization. The `.Rprofile` will set up R as a non-interactive `foreach` worker, which was already done at pod creation time. At this point, we can close the ssh connection and let the node run non-interactively.

Monitor Analyses. To ensure that the computation is proceeding correctly, we can monitor the node's console output with kubectl logs, as in line 62. This command works like `tail -f`, that is, any new output from the pod is continuously transmitted to the local host and output to the console. No errors should be observed and diagnostic output about the current voxel should exhibit a gradual increase. Once it is confirmed that things are proceeding smoothly then the parallelism can be increased by resizing the cluster (Figures 7 and 8). The resizing changes the number of CPUs available.

```
63 kubectl scale --replicas=32 rc/rworker-rc
64 kubectl cp $wk:/root/LGCout.txt LGCout.txt
```

Once the cluster is resized, we tell Kubernetes to replicate more worker pods (line 63). New nodes automatically add their muscle to the `foreach` loop via `Redis`. The number of replicas (32 in line 63) should match the number of CPUs available in the cluster. The first worker that manages execution of the `foreach` loop consumes negligible resources and can be ignored. We copy results back to our local machine every so often

(line 64). Cloud machines are usually reliable, but hardware can fail. Copying results periodically will ensure that it is relatively painless to resume after interruption.

For large, expensive jobs, money may not be available to run the job more than once. If results are corrupted by some bug hiding in the code, the money may be completely wasted. One way to guard against this troubling outcome is to run a small subset of the analyses both locally and on the cloud and compare whether the fit values and estimates match. If correlations are not above .99 then it would be wise to track down the cause of the differences before proceeding with the rest of the dataset.

As soon as all the computations are complete, we copy the final results (line 64) and then ensure that the cluster is shutdown to stop the billing clock (Figures 9 and 10). There is no persistent storage so any results not copied from the cluster are lost.

Results

The full latent growth curve model contained thirteen variance components and four means parameter estimates; five submodels were fit, each removing between five and ten variance components. The original analyses were performed on a 3.7 GHz Quad Core Mac Pro Desktop computer entirely dedicated to the task, requiring over four months of processing time to iterate over 40,962 vertices (contralateral only). The reanalysis was accomplished in 2 hr of real time. In both cases, parameter estimates were recorded to seven significant digits. Comparison of fit statistics demonstrated perfect agreement in -2 log likelihood for both the full model (Figure 11) and all submodels. Similarly, the seventeen maximum likelihood parameter estimates in the full model run in the cloud perfectly matched (to seven significant digits) those from the Mac Pro for all vertices.

Discussion

Cloud computing is a promising service for data analyses because you only pay for what you use. Here we detailed the assembly of a set of software solutions, `Redis`, Docker, and Kubernetes, which simplify the execution of parallel jobs in `R`, and demonstrated how

to analyze an MRI data set. Little software development was required because of the utilization of off-the-shelf software components.

Here we made the expedient assumption that all voxels are independent. Since this assumption can lead to absurd results (e.g., Bennett, Miller, & Wolford, 2009), it may be desirable to control the false discovery rate (e.g., Benjamini, 2010). Another solution is to model the dependence between voxels using a multilevel approach (e.g., Chen et al., 2018; Gelman, Hill, & Yajima, 2012). However, with the multilevel approach, it is not so simple to benefit from massively parallel processing as it is with independent models. One approach is to allocate more than one core to single workers. Using 96 threads on a symmetric multiprocessing node (the current maximum on Google Cloud) could yield substantial performance benefits. Other models may be constrained by memory demands instead of CPU. One example is a structural equation model that involves a genetic relatedness matrix. In these kinds of analyses, the relatedness between every individual in a large sample ($N > 5000$) is used to estimate the effects of genotype on traits of interest. Evaluating the likelihood during optimization thus involves inverting matrices of dimension as large as the sample size, which is computationally intensive. Another application would be analyses of nationwide population data, where sample size runs into the millions, and full information maximum likelihood is needed to handle all the missing data patterns. Large memory nodes (up to 1.4 TiB on Google Cloud) can be used to accommodate the matrices involved. However, for embarrassingly parallel applications, such as the MRI data used here, or for simulation studies, using many processors with relatively little memory is the optimal strategy.

Innovation in cloud computing is rapid. In this article, we carefully selected the most popular and vigorously maintained software. However, the details presented here could become outdated within 10 years. For example, the Docker container format faces competition from Rocket (https://coreos.com/rkt/). Since Rocket is more closely tailored to the needs of Kubernetes, Rocket may displace Docker in the future. Some further

research will be needed to determine how to best deploy parallel structural equation modeling should new tools become available.

The future is bright for inexpensive processing power. While large, well funded institutions may continue to maintain their own high performance computing clusters, researchers now have the option of renting similar hardware on an as-needed basis. As computer technology advances and the cost of cloud computing gradually diminishes, innovation in the sophisticated modeling of large datasets will likely be spurred by the unprecedented compute power available for rent.

References

Analytics, R. & Weston, S. (2015). *foreach: Provides foreach looping construct for R*. R package version 1.4.3. Retrieved from https://CRAN.R-project.org/package=foreach

Benjamini, Y. (2010). Discovering the false discovery rate. *Journal of the Royal Statistical Society: series B (statistical methodology)*, *72*(4), 405–416. doi:10.1111/j.1467-9868.2010.00746.x

Bennett, C. M., Miller, M., & Wolford, G. (2009). Neural correlates of interspecies perspective taking in the post-mortem atlantic salmon: An argument for multiple comparisons correction. *Neuroimage*, *47*(Suppl 1), S125.

Chen, G., Xiao, Y., Taylor, P. A., Rajendra, J. K., Riggins, T., Geng, F., ... Cox, R. W. (2018). Handling multiplicity in neuroimaging through Bayesian lenses with multilevel modeling. *bioRxiv*, 238998. doi:10.1101/238998

Cook, T. R. (n.d.). *rdockerHPC*. Retrieved March 6, 2018 from https://github.com/trcook/rdockerHPC. doi:10.5281/zenodo.1189980

Duncan, T. E. & Duncan, S. C. (2004). An introduction to latent growth curve modeling. *Behavior Therapy*, *35*(2), 333–363. doi:10.1016/S0005-7894(04)80042-X

Gelman, A., Hill, J., & Yajima, M. (2012). Why we (usually) don't have to worry about multiple comparisons. *Journal of Research on Educational Effectiveness*, *5*(2), 189–211. doi:10.1080/19345747.2011.618213

Ha, N.-T., Freytag, S., & Bickeboeller, H. (2014). Coverage and efficiency in current SNP chips. *European Journal of Human Genetics*, *22*(9), 1124. doi:10.1038/ejhg.2013.304

Kenny, S., Andric, M., Boker, S. M., Neale, M. C., Wilde, M., & Small, S. L. (2009). Parallel workflows for data-driven structural equation modeling in functional neuroimaging. *Frontiers in Neuroinformatics*, *3*, 34. doi:10.3389/neuro.11.034.2009

Lewis, B. W. (2015). *doRedis: "Foreach" parallel adapter using the "Redis" database*. R package version 1.2.0.

Luenberger, D. G. & Ye, Y. (2008). *Linear and nonlinear programming*. Springer-Verlag.

Neale, M. C. & Cardon, L. R. (1992). *Methodology for genetic studies of twins and families, NATO ASI series*. Dordrecht, The Netherlands: Kluwer Academic Press.

Neale, M. C., Hunter, M. D., Pritikin, J. N., Zahery, M., Brick, T. R., Kirkpatrick, R., ... Boker, S. M. (2016). `OpenMx` 2.0: Extended structural equation and statistical modeling. *Psychometrika*, *81*(2), 535–549. doi:10.1007/s11336-014-9435-8

Plöthner, M., Frank, M., & von der Schulenburg, J.-M. G. (2017). Cost analysis of whole genome sequencing in German clinical practice. *The European Journal of Health Economics*, *18*(5), 623–633. doi:10.1007/s10198-016-0815-0

Schmitt, J. E., Neale, M. C., Fassassi, B., Perez, J., Lenroot, R. K., Wells, E. M., & Giedd, J. N. (2014). The dynamic role of genetics on cortical patterning during childhood and adolescence. *Proceedings of the National Academy of Sciences*, *111*(18), 6774–6779. doi:10.1073/pnas.1311630111

Uğurbil, K., Xu, J., Auerbach, E. J., Moeller, S., Vu, A. T., Duarte-Carvajalino, J. M., ... Yacoub, E. (2013). Pushing spatial and temporal resolution for functional and diffusion MRI in the Human Connectome Project. *NeuroImage*, *80*, 80–104. Mapping the Connectome. doi:10.1016/j.neuroimage.2013.05.012

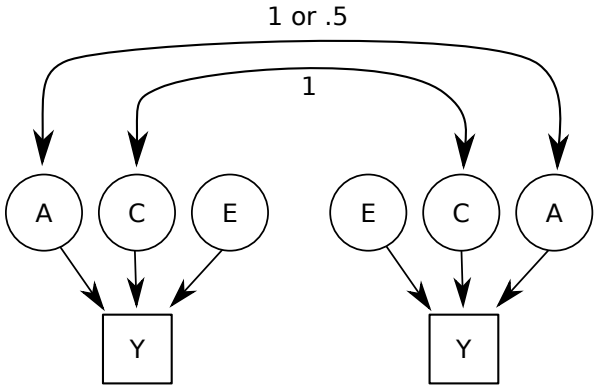

Figure 1. Path diagram for univariate twin model.

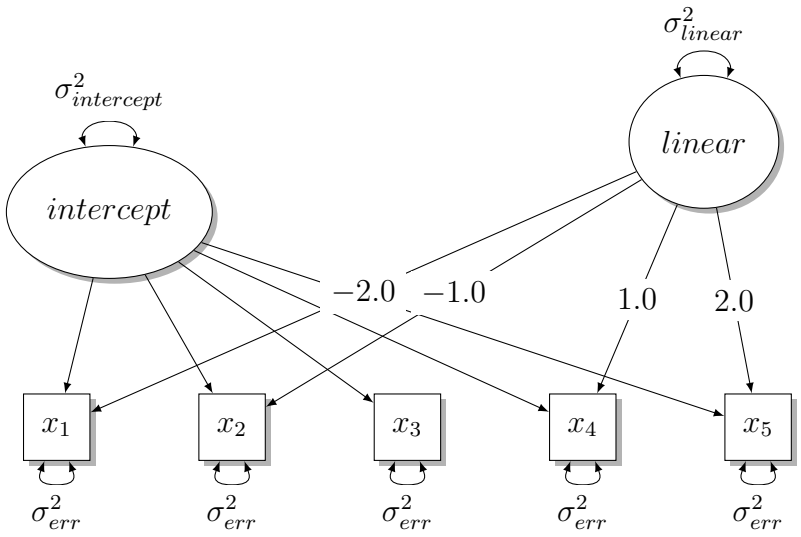

Figure 2. Path diagram for latent growth curve model.

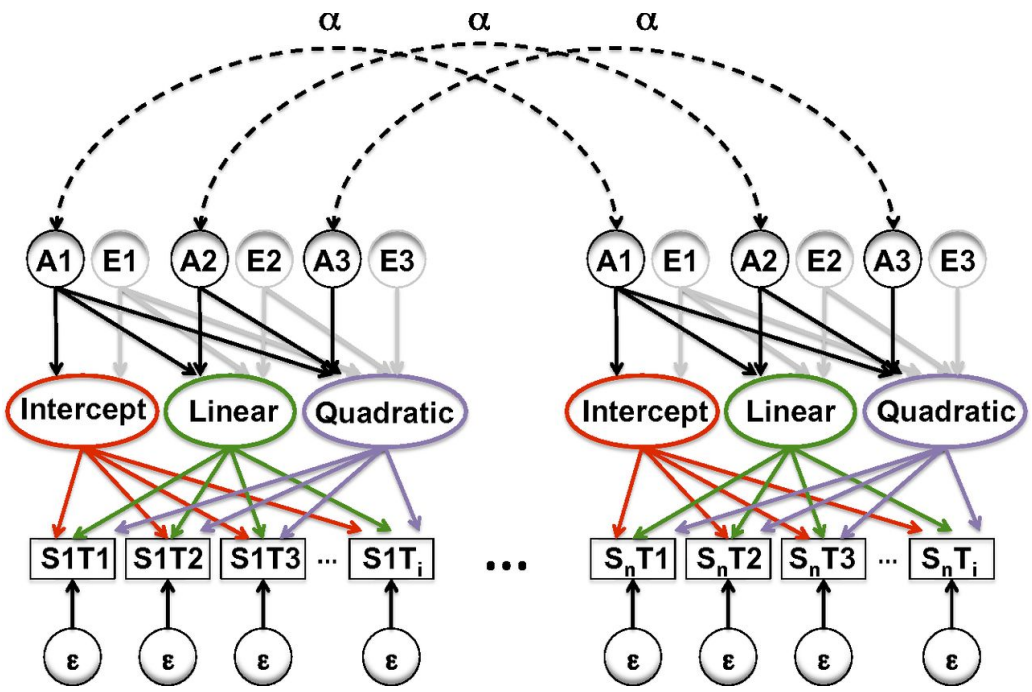

Figure 3. Path diagram of the genetically informative latent growth curve model.

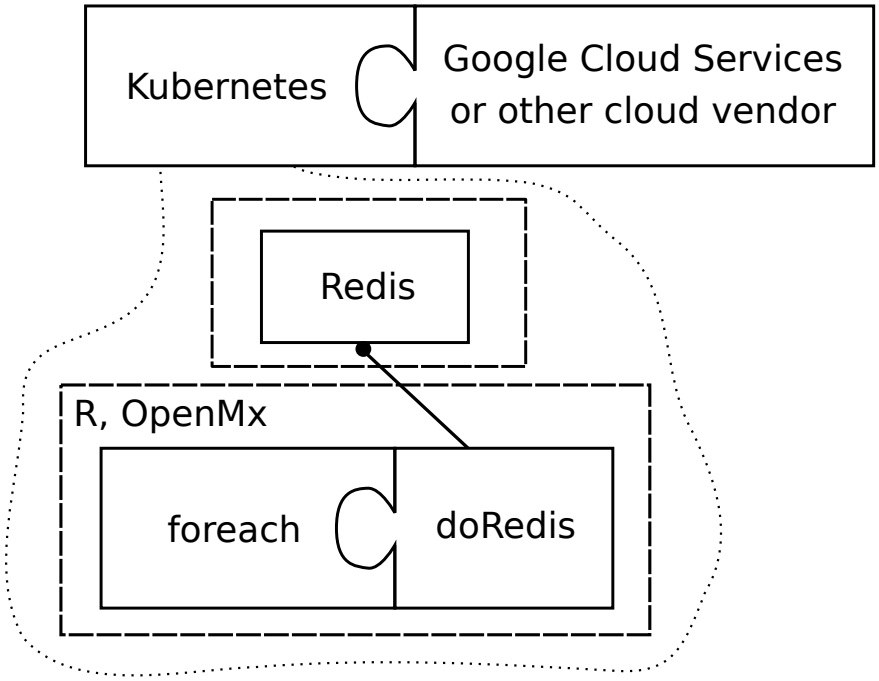

Figure 4. Architecture of a massively parallel cloud computing cluster. Kubernetes provides a vendor-neutral interface to cloud services. Within a Kubernetes cluster (dotted region), there is a single Redis key-value database that communicates with many instances of R via the `doRedis` plugin. The `doRedis` module provides an implementation of the `foreach` interface. There are two Docker images (dashed lines), a standard image for `Redis` and the custom image that contains the scripts, data, and R packages for a particular analysis. Docker images also contain sufficient components of the Linux operating system for self-contained operation. Kubernetes communicates with a cloud service provider to manage the creation and destruction of Docker images.

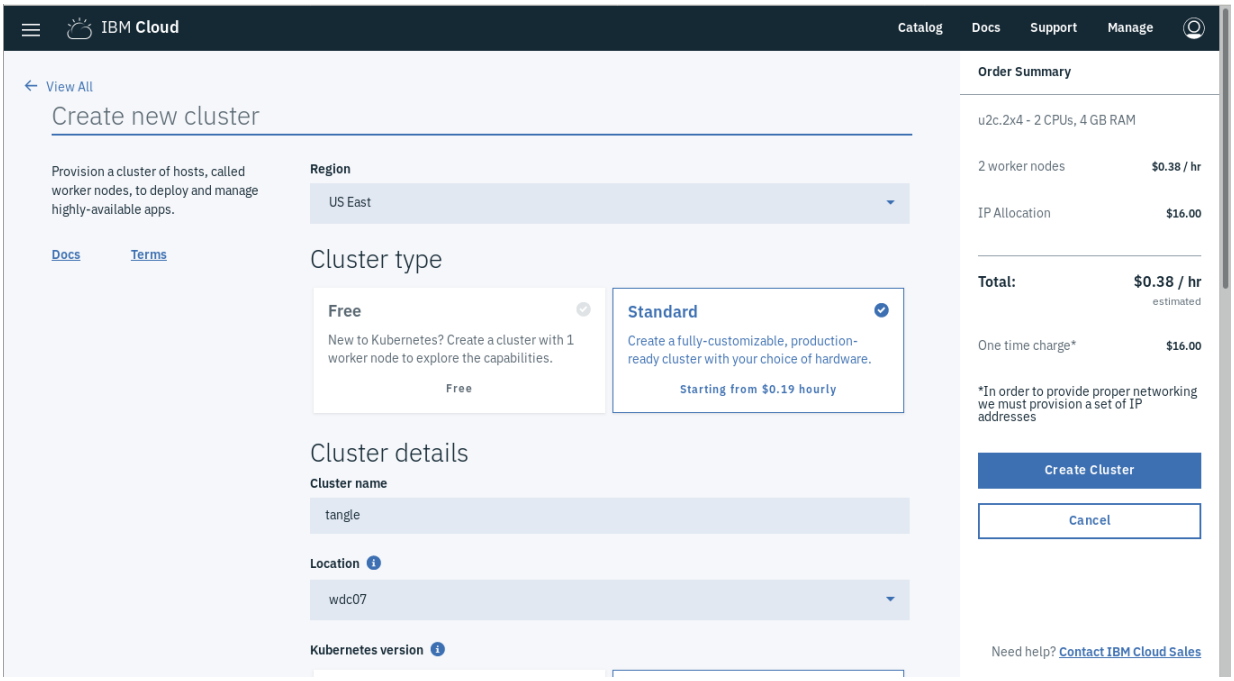

Figure 5. Create a cluster called *tangle* in BlueMix.

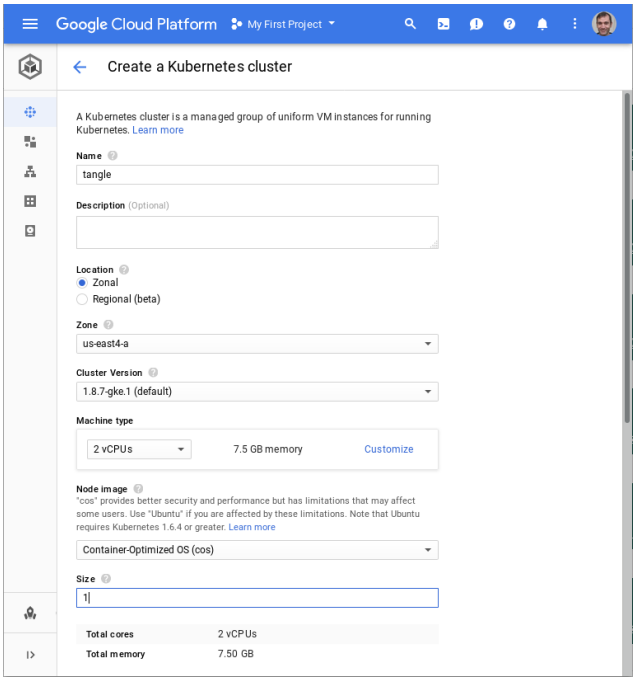

Figure 6. Create a cluster called *tangle* in Google Cloud.

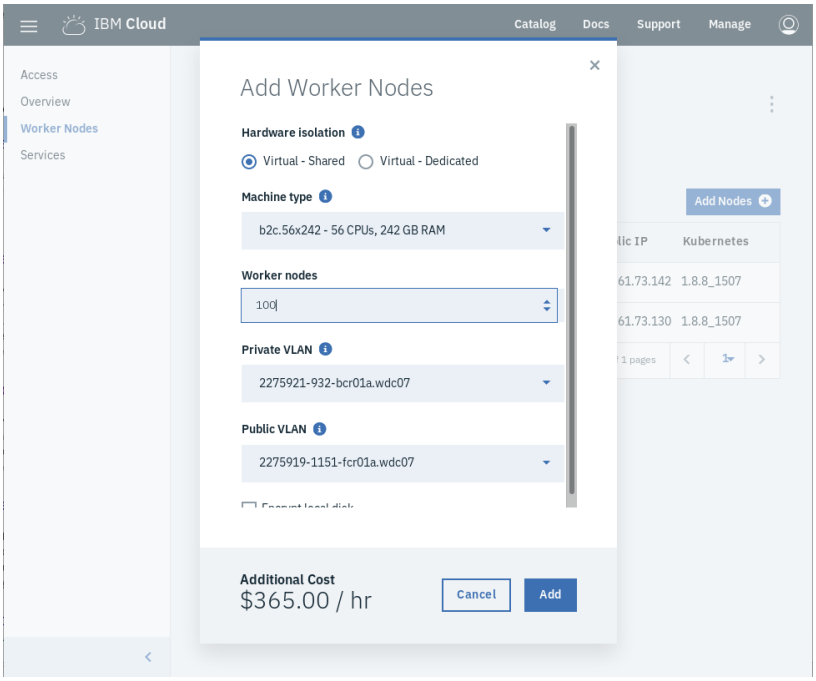

Figure 7. Add more nodes in BlueMix.

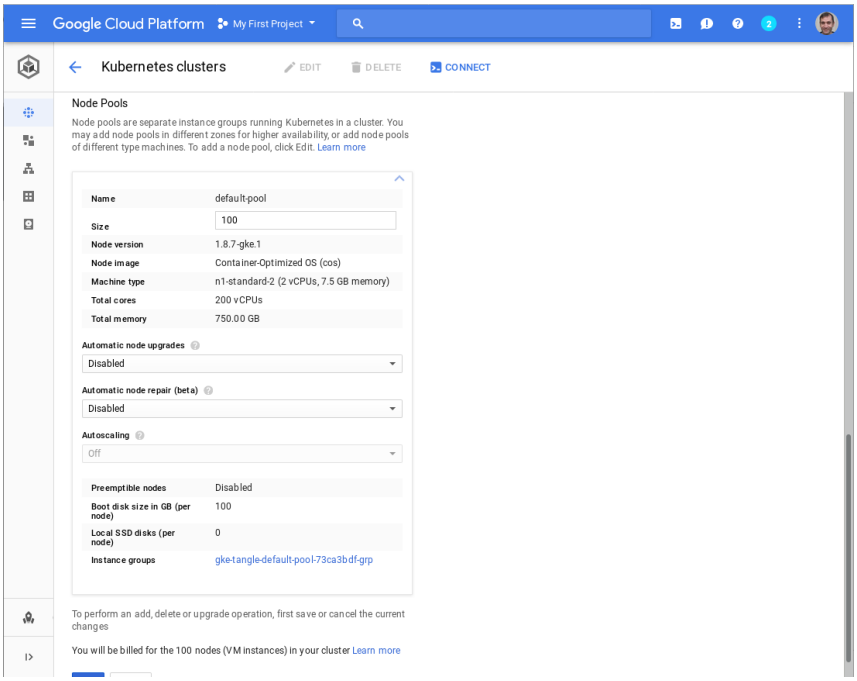

Figure 8. Add more nodes in Google Cloud.

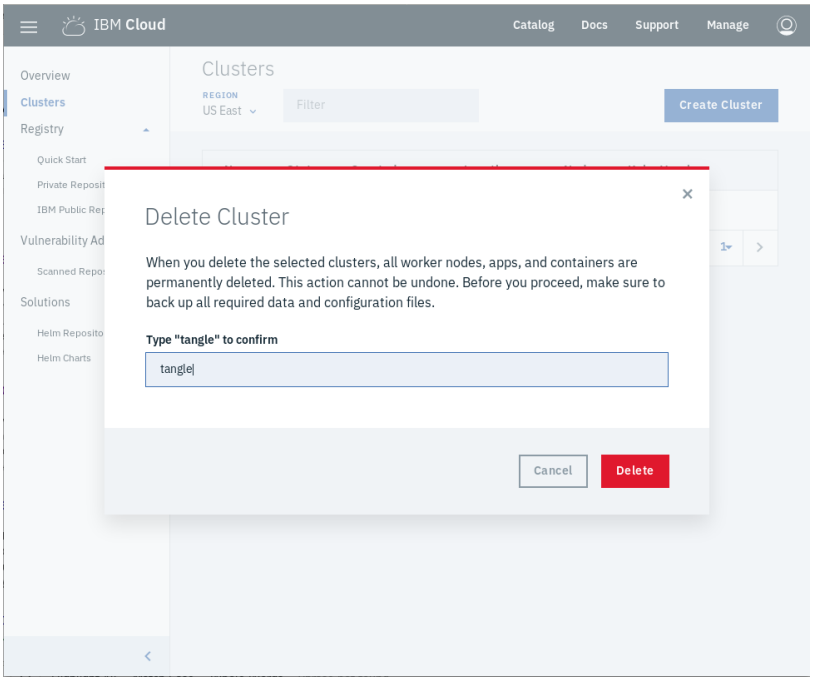

Figure 9. Shutdown the cluster in BlueMix.

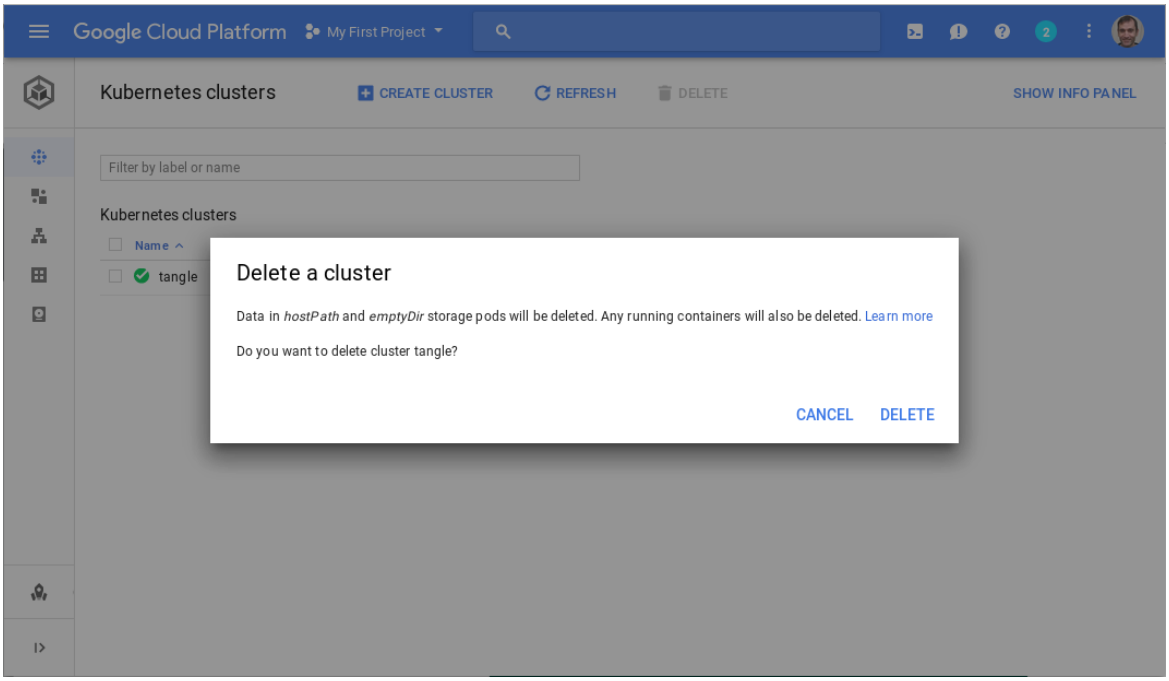

Figure 10. Shutdown the cluster in Google Cloud.

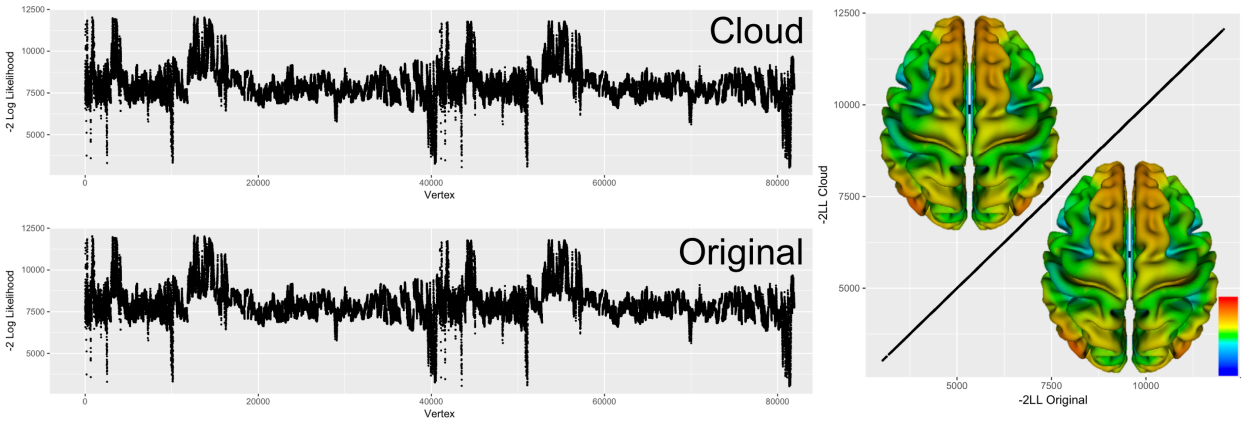

Figure 11. Fit statistics for both original and cloud analyses, by vertex ID. −2LL was perfectly correlated, to seven significant digits, producing identical brain maps.

Appendix A

Worker Rprofile

```
options('redis:num'=TRUE)
require(doRedis)
library(base)
library(stats)
.First <- function(){
  library(doRedis)
  redisWorker(host='redis',queue='jobs',loglevel=1)
  print("DONE")
}
```

Appendix B

Redis Pod Specifications

The file `redis.yaml` contains:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-svc
  labels:
    name: redis-svc
spec:
  replicas: 1
  selector:
    name: redis-svc
  template:
    metadata:
      labels:
        name: redis-svc
    spec:
      containers:
      - name: redis
        image: redis
        ports:
        - containerPort: 6379
```

The file `redis_service.yaml` contains:

```
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    name: redis-svc
spec:
```

```
8     ports:
9     - port: 6379
10      targetPort: 6379
11    selector:
12      name: redis-svc
```