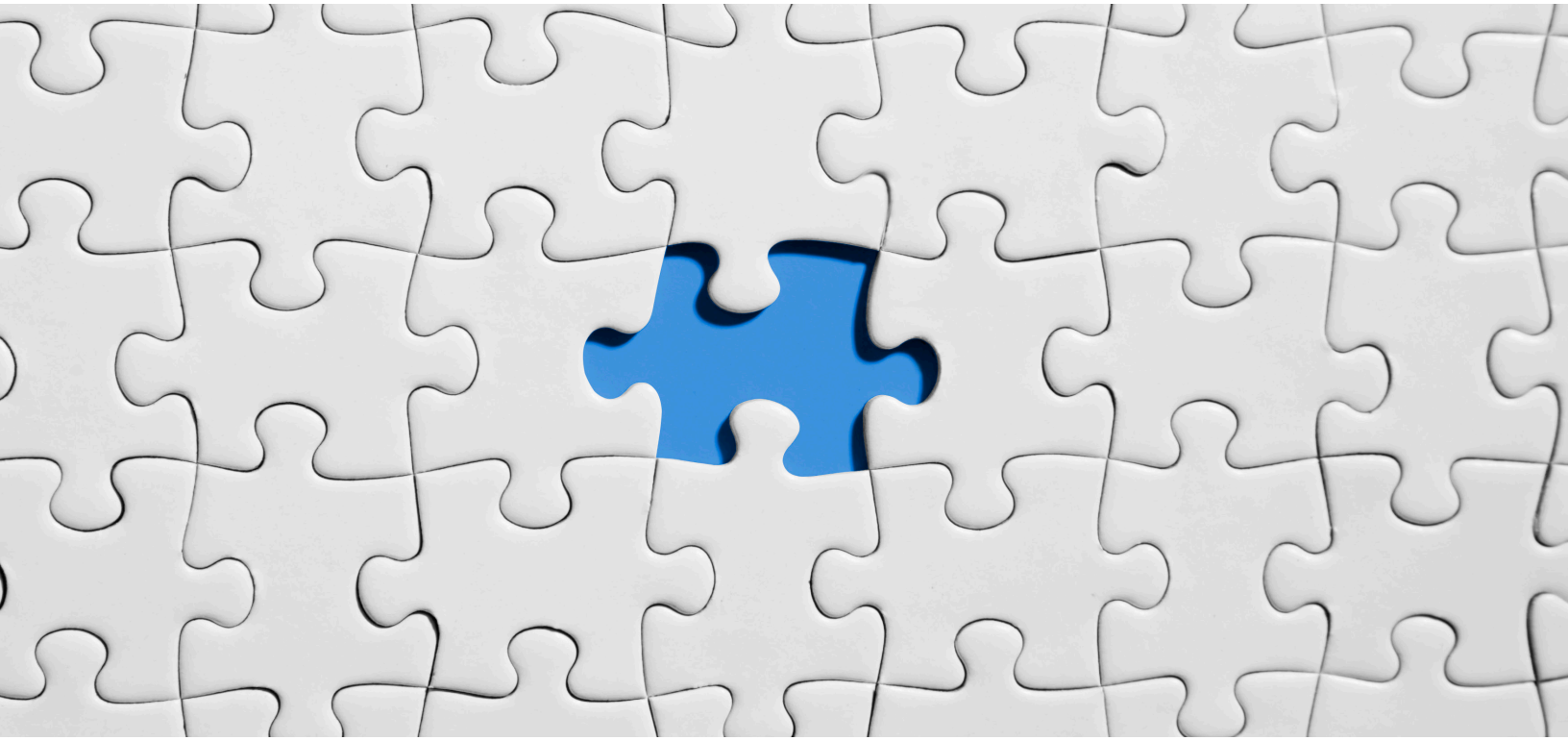
DELL Technologies

AUTODISCOVERY PROXY TO PROTECT CLOUD RESOURCES



Pablo Calvo
Delivery Specialist
Dell Technologies
Pablo.calvo@dell.com

Claudio Jeniec
Data Protection & Storage Engineer
Uniqs S.A.
Claudio.jeniec@uniqs.com.ar

Rex Torti
Cloud & Infrastructure Engineer
Uniqs S.A.
Rex.torti@uniqs.com.ar

Norman Arteaga
DevOps Implementation Specialist
Uniqs S.A.
Norman.arteaga@uniqs.com.ar

DELL Technologies Proven Professional

Knowledge Sharing Article

The Dell Technologies Proven Professional Certification program validates a wide range of skills and competencies across multiple technologies and products.

From Associate, entry-level courses to Expert-level, experience-based exams, all professionals in or looking to begin a career in IT benefit from industry-leading training and certification paths from one of the world's most trusted technology partners.

Proven Professional certifications include:

- Cloud
- Converged/Hyperconverged Infrastructure
- Data Protection
- Data Science
- Networking
- Security
- Servers
- Storage
- Enterprise Architect

Courses are offered to meet different learning styles and schedules, including self-paced On Demand, remote-based Virtual Instructor-Led and in-person Classrooms.

Whether you are an experienced IT professional or just getting started, Dell Technologies Proven Professional certifications are designed to clearly signal proficiency to colleagues and employers.

Learn more at www.dell.com/certification

Table of Contents

Introduction

Data protection in the cloud is the practice that has become increasingly challenging as more companies move to the cloud. The challenge is more complex than ever, as data may be geographically dispersed over multiple data centers in public, private and hybrid cloud environments as well as on-premises locations.

This article shows how to use Dell Technologies data protection products to protect any set of cloud data, discovering them dynamically, "Autodiscovery Proxy To Protect Cloud Resources" is an effective, integrable, easy to deploy and low cost solution that covers the gap in current cloud backup solutions. It enables us to recognize tagged objects and learn which of them need some kind of protection.

Our solution uses a *dynamic discovery capability,* which allows learning which cloud resource objects need to be backed up. The solution extracts the objects to be protected from a kind of inventory learned automatically.

The lifetime of the backed up data can be configured by backup policies defined according to the chosen backup product, including schedules and administration of long-term retention policies, data access profiles, lock retention, tiering, deduplication, etc. External schedulers such as Azure function[1] or AWS Lambda services[2], among others, can also be used to launch tasks and perform job controls.

Autodiscovery Proxy To Protect Cloud Resources is implemented in a container technology, taking advantage its portability, scaling and repeatability and can deployed by DevOps orchestration tools over Kubernetes[3] or others. With this architecture, the solution scenario can be designed with high availability considerations.

This thin container architecture has a minimal total cost of ownership (TCO). It requires very few computing, network and storage resources due to the fact that the data is stored in the pre-existing backup devices; it can even be turned off when not in use to further optimize costs. Once the container image is created, it can be ported effortlessly in multi-tenant environments.

This proxy only requires a configuration file – just one file! - and allows automated deployment in minutes since it can be integrated with any orchestration tool. As this is considered another client for backup applications, it can be integrated into the flow of administration and control of protection tasks.

Different applications can be protected within the same container called a “multi-purpose” container. This feature is very useful when the cloud environment is small and does not require much processing demand.

In the case of advanced configurations, it is feasible to balance the load of the same type of backup in more than one container to parallelize the sending of the protected data.

As the backup data is stored in Dell Technologies products, i.e. Avamar, NetWorker, DataDomain, among others, all the options that govern their replication capabilities between regions, availability zones and hybrid environments, build simpler, faster and more cost-efficient disaster and data recovery.

Security has not been neglected. If passwords are required to access objects that need protection, or any other key/value combination, they can be stored and queried from standard

[1] Azure Functions is an event driven, compute-on-demand experience that extends the existing Azure application platform with capabilities to implement code triggered by events occurring in Azure or third party service as well as on-premises systems (https://azure.microsoft.com/en-us/blog/introducing-azure-functions/)

[2] AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers (https://aws.amazon.com/lambda/)

[3] Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications (https://kubernetes.io/)

system vaults (such as Azure Key Vault[4] and AWS KMS[5], among others) avoiding exposure of the requested values. It is recommended to generate read-only keys with password rotation.

With our Autodiscovery Proxy To Protect Cloud Resources we have solved demands for dynamic discovery and backup of various data sources that DevOps environments require using all the strengths of current backup products.

The backup administrator no longer has to worry about how to ensure data protection every time new services are configured and started because Autodiscovery Proxy does it in an automated and efficient way.

It can be deployed to any cloud provider that certifies Dell Technologies backup products (e.g. NetWorker, Avamar, Data Domain), including Azure, AWS, and GCP.

Please see reference #1 in the Reference section to see a complete demonstration of this deployment.

How to build and deploy

Figure 1 depicts the components of the solution.

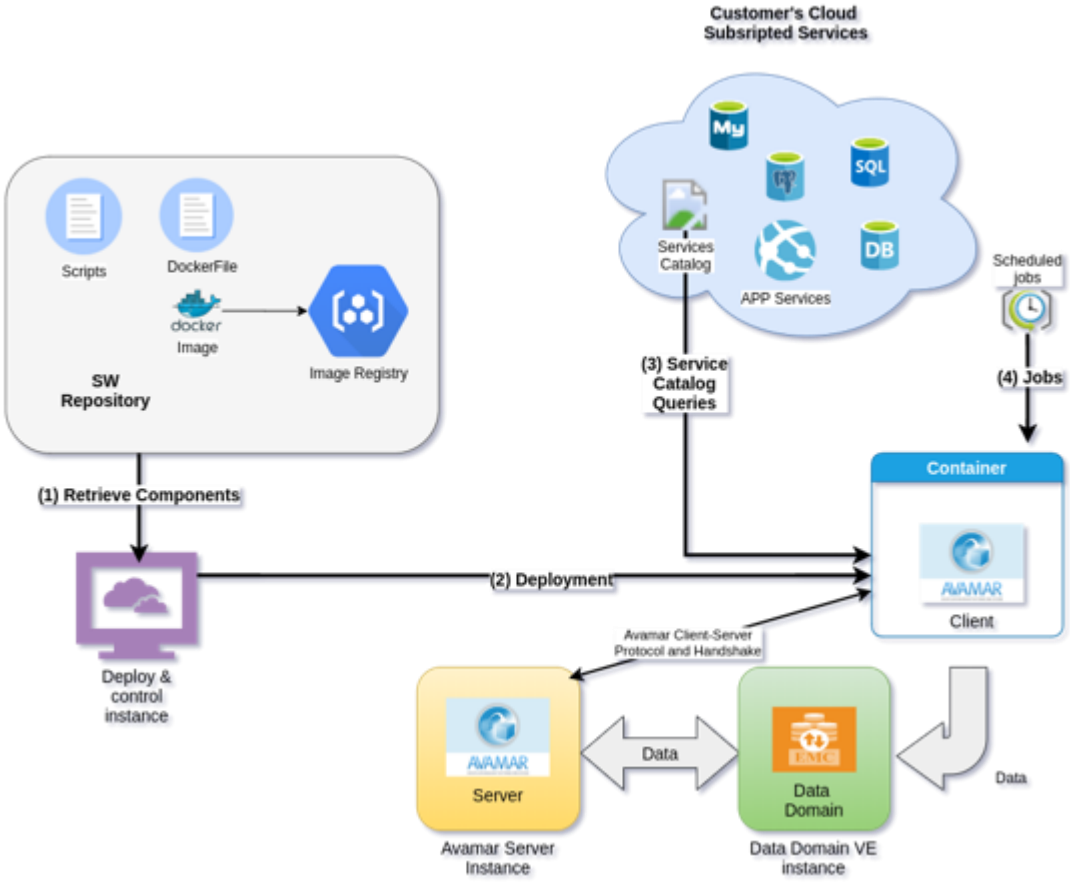


Figure 1 - Deployment flow

[4] Azure Key Vault is a cloud service for securely storing and accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, certificates, or cryptographic keys. (https://docs.microsoft.com/en-us/azure/key-vault/general/basic-concepts)

[5] AWS Key Management Service (KMS) makes it easy for you to create and manage cryptographic keys and control their use across a wide range of AWS services and in your applications (https://aws.amazon.com/kms/)

Deploy Control Instance (DCI)

DCI is the workspace used to configure images, launch, test, and distribute Docker (or Podman) containers; it is not a production environment. CPU, memory, and disk space values can be modified according to sizing; Figure 2 shows a Standard B2s. You can choose the version of Linux that best fits your needs. Here, we are using Red Hat Linux 8.2.

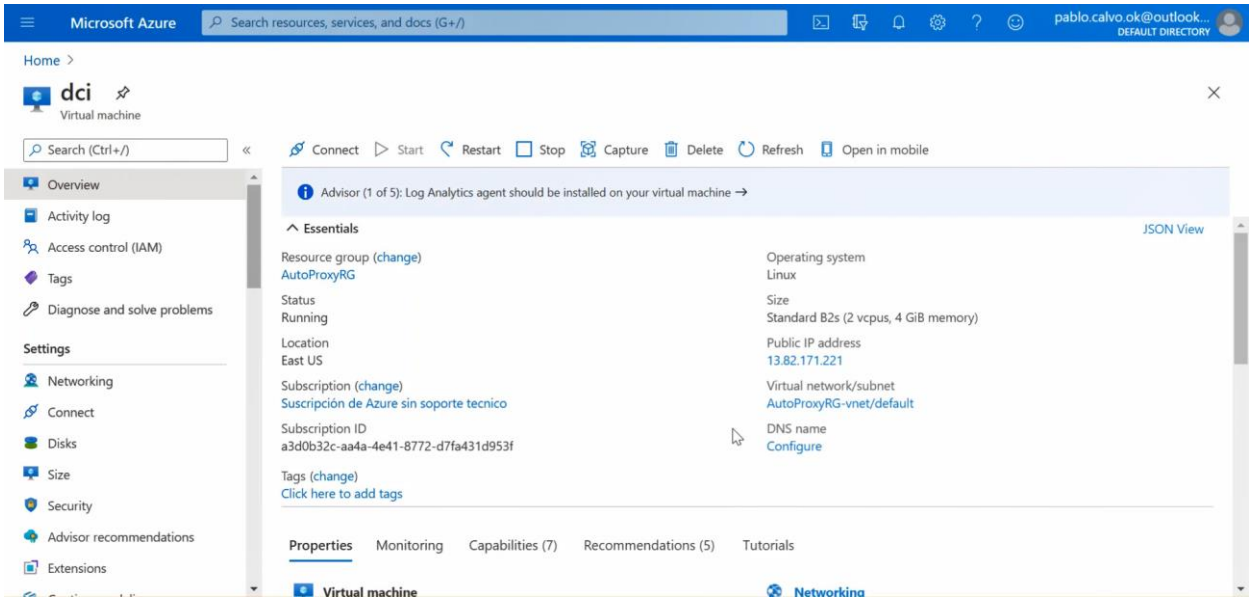


Figure 2 - Azure DCI virtual machine

The solution needs a virtual environment that will run a number of containers specialized to back up and protect different database services (as mentioned earlier)

SW repository

This project uses the following technologies:

- Docker [6] (or Podman[7]) config files called "dockerfiles"
- JSON files[8] with .json extension
- Shell scripts with .sh extension
- ReadMe files with .md extension
- Avamar clients with .rpm extension, DDBoost File System (DDBoostFS) with .rpm extension, PostgreSQL[9] with .rpm extension
- Azure command line -Azure CLI-[10] installed from the azure-cli.repo file.
- Certificates to connect to the cloud provider (PEM files with extension .pem[11])

[6] Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly (https://docs.docker.com/get-started/overview/)

[7] Podman is a daemonless, open source, Linux native tool designed to make it easy to find, run, build, share and deploy applications using Open Containers Initiative (OCI) Containers and Container Images. (https://podman.readthedocs.io/en/latest/index.html)

[8] JSON (JavaScript Object Notation) is a lightweight data-interchange format (https://www.json.org/json-en.htm l)

[9] PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads (https://www.postgresql.org/about/)

[10] The Azure command-line interface (Azure CLI) is a set of commands used to create and manage Azure resources (https://docs.microsoft.com/en-us/cli/azure/what-is-azure-cli)

[11] Privacy Enhanced Mail (PEM) files are a type of Public Key Infrastructure (PKI) file used for keys and certificates (

The content can be downloaded from a public Github repository using standard methods. Its URL is https://github.com/uniqs-devops/bkp-proxy.git

Repo layout:

```
├── avamar.19.2-postgresql-azure.dockerfile
├── avamar-PG-Azure-template-Avamar.dockerfile
├── avamar-PG-Azure-template-DDBoostFS.dockerfile
├── avamar-PG-Azure-template.dockerfile
├── dps-setup.json
├── dps-setup.sh
├── README1st.md
├── README-dps-setup.json.md
├── README-dps-setup.md
└── src
    ├── avamar
    │   ├── backup-postgreSQL.sh
    │   ├── resources
    │   └── setup.sh
    ├── azure
    │   ├── azure-cli.repo
    │   └── azurelogin.pem
    ├── ddboostfs
    │   └── boostfs.lockbox
    └── packages
        ├── AvamarServerPackages
        │   └── 19.2
        │       └── Dummy package.avp
        └── DockerEmbebed
            ├── 19.2
            │   ├── AvamarClient-linux-sles11-x86_64-19.2.100-155.rpm
            │   └── DDBoostFS-7.2.0.5-654559.rhel.x86_64.rpm
            ├── 19.3
            │   ├── AvamarClient-linux-sles11-x86_64-19.3.100-149.rpm
            │   └── DDBoostFS-7.2.0.5-654559.rhel.x86_64.rpm
            └── postgresql
                └── pgdg-redhat-repo-latest.noarch.rpm
```

dps-setup.json is the unique configuration file, see an example below:

```
{

 "cloudProvider": "Azure",
 "dockerType": "postgresql",
 "dockerTypeName": "PG",
 "keyVaultName": "keyvault1414",
 "tenantId": "054bb9ef-86b1-4f4e-a843-deb19d532c11",
 "avamar": {
   "useAvamar": "YES",
   "avamarServerName": "avedemo01.internal.cloudapp.net",
      "avamarDomain": "clients",
      "installDir": "dockerclient",
      "avamarVersion": "19.3"

 },
 "datadomain": {
   "datadomainServerName": "ddvedemo01.internal.cloudapp.net",
      "mountType": "DDBoostFS",
```

https://s3.amazonaws.com/smhelpcenter/smhelp941/classic/Content/security/concepts/what_are_pem_files.htm)

```
      "RootBackupDir": "Backup"
  },
  "ddboosfs": {
    "storageUnit": "st-ddboostfs"
  },
  "container": {
    "containerName": "dockerpg-01"
  },
  "azureResources": [
    {
      "type": "PG",
      "resourceType": "Microsoft.DBforPostgreSQL/servers"
    },
    {
      "type": "PG",
      "resourceType": null
    },
    {
      "type": "PG",
      "resourceType": null
    }
  ],
  "backupTags": [
    {
      "type": "user",
      "value": "bck_user"
    },
    {
      "type": "port",
      "value": "bck_port"
    },
    {
      "type": "server",
      "value": "bck_server"
    },
    {
      "type": "database",
      "value": "bck_database"
    },
    {
      "type": "task",
      "value": "bck_task"
    },
    {
      "type": "secret",
      "value": "bck_secret"
    }
  ]
}
```

json file description:

- cloudProvider
``` Azure or AWS or GCP.```
- dockerType
``` PostgreSQL or CosmoDB or resource type to backup.```
- dockerTypeName
``` Sort of dockerType```
- keyVaultName
``` Azure Kay Vault name```
- tenantId
``` Tenantid```
- useAvamar
``` Use avamar to store backup data```
- avamarServerName and datadomainServerName
``` Avamar and Data Domain FQDN.```
- avamarDomain
``` Avamar docker domain, eg. clients```
- avamarVersion
``` Avamar version```
- mountType
``` DDBoostFS or NFS```
- RootBackupDir
``` DDBoostFS or NFS mount point on container```
- storageUnit
``` Data Domain Storage Unit used to hold data```
- containerName
``` FQDN of container used to register this client on Avamar.
    Add forward and reverse DNS records to DNS Server```
- resourceType
``` Azure resource type to be discovered```

- backupTags \ Type
``` Type of tag```
- backupTags \ Value
``` Value of type tag```

## Dockerfile example

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image[12]

### Sections

Dependencies and basic software install section.

- Cronie package is used if crontab is needed.
- jq[13] package is used to process json configuration files.

```
#!/bin/sh
FROM centos:latest
# Install SO packages
RUN yum install -y --setopt=tsflags=nodocs openssh-server \
 && yum install -y --setopt=tsflags=nodocs iproute net-tools initscripts \
 && yum install -y --setopt=tsflags=nodocs jq cronie\
 && yum clean all
workdir /tmp
RUN mkdir /dockerclient
```

There are some applications to be deployed in the DCI in this scenario, for example:

- Installs PostgreSQL client
- Copies backup-postgreSQL.sh backup script (executes the pg_dump command)
- Installs AZ CLI and copy JSON file
- Installs DDBoostFS client.
- Installs Avamar client package (rpm in this example).
- Copies .avagent file to fix container hostname (Avamar client name).

```
# Copy PosgreSQL repo install package
COPY src/packages/DockerEmbebed/postgresql/pgdg-redhat-repo-latest.noarch.rpm /tmp
# Install PosgreSQL client & repo package
RUN yum install -y /tmp/pgdg-redhat-repo-latest.noarch.rpm
RUN yum install -y postgresql
# Copy  backup script
COPY src/avamar/backup-postgreSQL.sh /dockerclient
RUN chmod 755 /dockerclient/backup-postgreSQL.sh
# Copy .pem file
COPY src/azure/azurelogin.pem /dockerclient
# Install AZ CLI
RUN rpm --import https://packages.microsoft.com/keys/microsoft.asc
COPY src/azure/azure-cli.repo /etc/yum.repos.d
RUN yum install -y azure-cli
# json file
COPY dps-setup.json /dockerclient
# Open the SSH port
COPY src/packages/DockerEmbebed/19.2/DDBoostFS*.rpm /tmp
# Install DDBoostFS
RUN yum localinstall -y /tmp/DDBoostFS*.rpm
# Copy DDBoostFS lockbox file
COPY src/ddboostfs/boostfs.lockbox /opt/emc/boostfs/lockbox/boostfs.lockbox
# Copy avamar Client to /tmp for installation
COPY src/packages/DockerEmbebed/19.2/AvamarClient-linux-sles11-x86_64-19.2.*.rpm /tmp
# Install avamar client usen RPM as Install Guide procedure
RUN rpm -ivh --relocate /usr/local/avamar=/dockerclient  /tmp/AvamarClient-linux-sles11-x86_64-19.2.*.rpm
#Copy .avagent file
COPY src/avamar/.avagent /dockerclient
```

Ports: We expose the necessary ports for Avamar client, register and start the agent.

[12] https://docs.docker.com/engine/reference/builder/

[13] jq is a lightweight and flexible command-line JSON processor (https://stedolan.github.io/jq/)

```
#open the SSH port
EXPOSE 22
#Avamar Client inbound ports
EXPOSE 28002
EXPOSE 30001
EXPOSE 30002
#Avamar Client outbound ports
EXPOSE 53
EXPOSE 123
EXPOSE 443
EXPOSE 3008
EXPOSE 8105
EXPOSE 8109
EXPOSE 8181
EXPOSE 8444
EXPOSE 27000
EXPOSE 27001
EXPOSE 29000
EXPOSE 30101
EXPOSE 30102
#auto start Avamar services
COPY src/avamar/setup.sh /dockerclient
RUN chmod 755 /dockerclient/setup.sh
RUN /dockerclient/setup.sh
```

We leave the agent up

```
# Cleanup /tmp folder, agent start  and Configuration persist
RUN rm -f /tmp/*.rpm
ENTRYPOINT mount -a &&  [ -f /etc/init.d/avagent ] && /etc/init.d/avagent start && /bin/bash
```

Deployment Steps

Steps # 1 and # 2 are manual configurations to download Project code and enable Data Domain access through DDboost File System. Step # 3 will be complete using the dps-setup.sh script.

1. Download code from GitHub repo[14]:

   sudo dnf install git
   git clone https://github.com/uniqs-devops/bkp-proxy.git

2. Requirements when use DDboostFS

   a. Create DD Boost user (Data Domain Admin Console - sysadmin access or similar is required)

      user add <DDBoost user> role user
      user password aging show
      user password aging set <DDBoost user> max-days-between-change 99999

   b. Create storage unit (Data Domain Admin Console - sysadmin access or similar is required)

      ddboost storage-unit create <storage-unit name> user <DDboost user>

   c. Install DDBoostFS in the Data Control Instance (DCI)

      sudo yum localinstall -y src/packages/DockerEmbebed/<version>/DDBoostFS-<version>.rhel.x86_64.rpm

      DDBoostFS is used to generate DDBoostFS lockbox.

[14] Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals (https://git-scm.com/docs/git)

d. Create a lockbox file (DCI side)

sudo /opt/emc/boostfs/bin/boostfs lockbox set -u <DDboost user> -d <Data Domain> -s <storage-unit>

3. Setup Procedure

a. Run dps-setup.sh -s to setup the DCI environment.

b. Run az ad sp create-for-rbac to create cert (.pem) file. Please run before 'az login' to set up account if you are not logged in yet.

az ad sp create-for-rbac --name 'PaaSBackup' --create-cert; mv ~/*.pem src/azure/azurelogin.pem

c. Run dps-setup.sh -p to prebuild dockerfile.

d. Run dps-setup.sh -b to create a new docker image.

e. Run dps-setup.sh -d <hostname> to deploy a new contanier.

f. Configure an Avamar policy backup as usual.

dps-setup.sh features:

- Install packages used by this DCI.
- Read json file keys to populate the dockerfile used to create docker images.
- Build a Docker image.
- Install Docker container on host, Kubernetes or Openshift

dps-setup.sh Usage:

dps-setup.sh -h

Please type '-s | --setup' to Setup or '-p | --prebuild' to Prebuild or '-b | --build' to Build or '-d | --deploy' to Deploy

Reference

```
-s | --setup                                  to setup environment
-p | --prebuild                          to complete files
-b | --build                               to create a docker images
-d | --deploy   --host  <hostname>     to run a new container on host
```

Service catalog queries

See sections “Automatic Discovery” and “Resource List (Tagged Values)”

Avamar Backup Jobs

In this example we will use Avamar as backup software to make our backups although we could, with a few modifications in the project, use NetWorker, NetBackup, Commvault or any other product.

To schedule a backup task, you need to create a Policy which will contain the following:

- Members: They are the resources to protect. Figure 3 depicts “dockerpg-01” which corresponds to the hostname of the Docker container that dumps the PostgreSQL database leaving its data in a Storage Unit of Data Domain using the DDboost File System (DDBoostFS) client.
- Dataset: See section “Backup catalog data”.
- Schedule: The execution plan of the backup task. External planners can also be used instead of Avamar to launch the job.
- Retention: Sets the lifetime of the information within the catalog. It is the period during which the backup records will exist to perform a restore of the information.

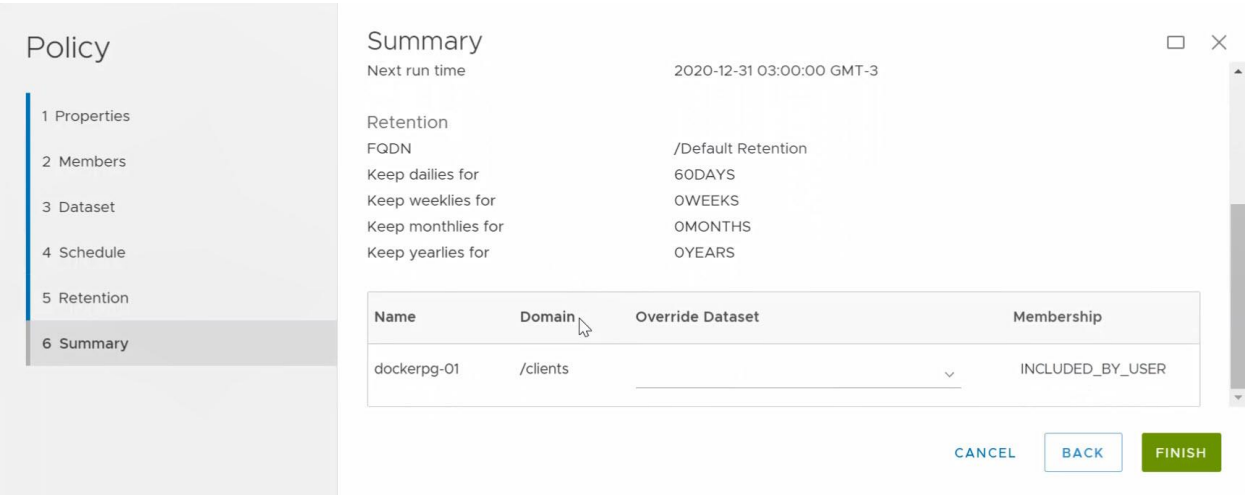


Figure 3 - Avamar Policy Summary

Architecture - How it Works (Azure use case)

Figure 4 shows the information flow of the whole solution when executing a PostgreSQL backup.

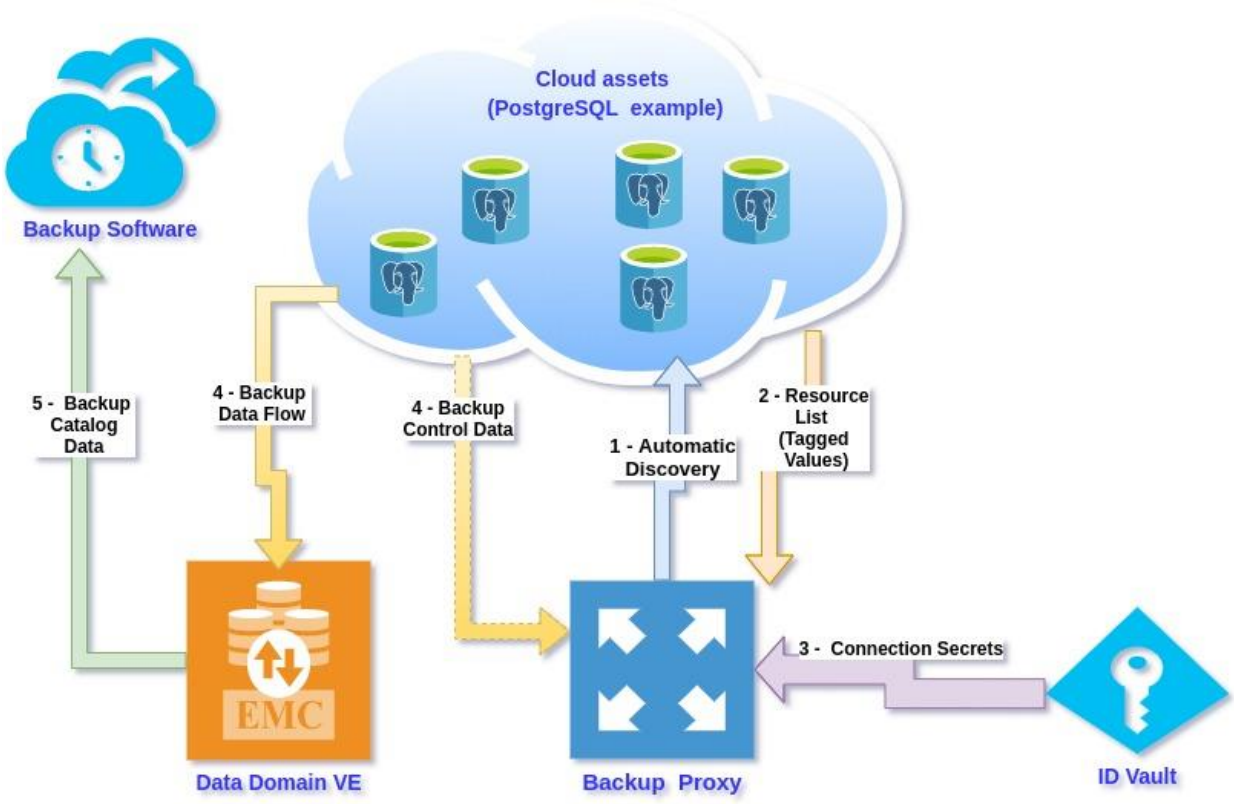


Figure 4 - How Autodiscovery Proxy to Protect Cloud Resources works

Steps

Automatic Discovery

Resource List (Tagged Values)

We are going to read the resourceType key inside azureResources to learn what kind of resources we'll search in the Azure subscription to which we have logged in. az resource list --resource-type <ResourceType>[15] is used to discover the Azure resources.

Once all resources were found we look for their tags[16] to obtain the properties required to perform the backup. If there are no tags no, there are no backups due to a lack of information.

Resource and tag names are "learned" from the JSON file configured at the beginning of the project.

All assets found and their properties will be stored together in the swoconfig.tmp file to be processed sequentially later.

```
RESOURCES=`jq '.azureResources[] | select(.type=="PG" and .resourceType != null)|.resourceType' \
$ConfigDir/dps-setup.json | sed 's/"//g'`
SERVICE_TYPE=`cat $ConfigDir/dps-setup.json  | jq -r '.dockerType'`
SERVER_TAG=`jq '.backupTags[] | select(.type=="server")|.value' $ConfigDir/dps-setup.json | sed 's/"//g'`
```

[15] List Azure resources (https://docs.microsoft.com/en-us/cli/azure/resource?view=azure-cli-latest#az_resource_list)

[16] Used to quickly locate resources associated with specific workloads, environments, ownership groups, or other important information (https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/decision-guides/resource-tagging/?toc=/azure/azure-resource-manager/management/toc.json)

```
DATABASE_TAG=`jq '.backupTags[] | select(.type=="database")|.value' $ConfigDir/dps-setup.json | sed 's/"//g'`
TASK_TAG=`jq '.backupTags[] | select(.type=="task")|.value' $ConfigDir/dps-setup.json | sed 's/"//g'`
SECRET_TAG=`jq '.backupTags[] | select(.type=="secret")|.value' $ConfigDir/dps-setup.json | sed 's/"//g'`


if [ -f ${ConfigDir}/swoconfig ]; then rm -rf ${ConfigDir}/swoconfig; fi
for resource in $RESOURCES
do
  az resource list  --resource-type $resource -o json > /tmp/az.json
  echo $SERVICE_TYPE >> ${ConfigDir}/swoconfig.tmp
  for i in $TASK_TAG $SERVER_TAG $DATABASE_TAG $USER_TAG $SECRET_TAG
  do
   jq '.[].tags' /tmp/az.json -r | grep $i | awk {'print $2'} | sed 's/"//g' | sed 's/,//g' \
   >> ${ConfigDir}/swoconfig.tmp
  done
  paste -sd " " ${ConfigDir}/swoconfig.tmp >  ${ConfigDir}/swoconfig; rm -f \
  ${ConfigDir}/swoconfig.tmp
done
 paste -sd " " ${ConfigDir}/swoconfig.tmp >  ${ConfigDir}/swoconfig; rm -f \
  ${ConfigDir}/swoconfig.tmp
done
```

Figure 5 shows an Azure PostgreSQL configuration resource called dbserver1414. Note that JSON keys are not needed since this resource is dynamically discovered.

Figure 6 shows the tags configured for Azure PostgreSQL. These keys correspond to the backup Tags and type/value entries of the JSON file.

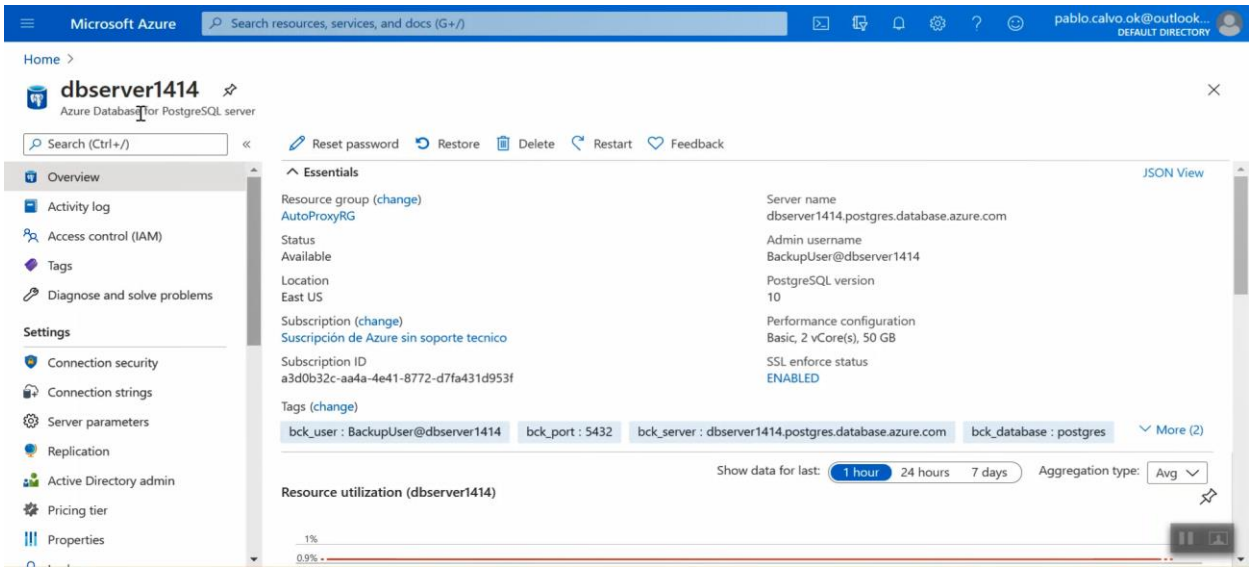


Figure 5 - Azure PostgreSQL configuration overview

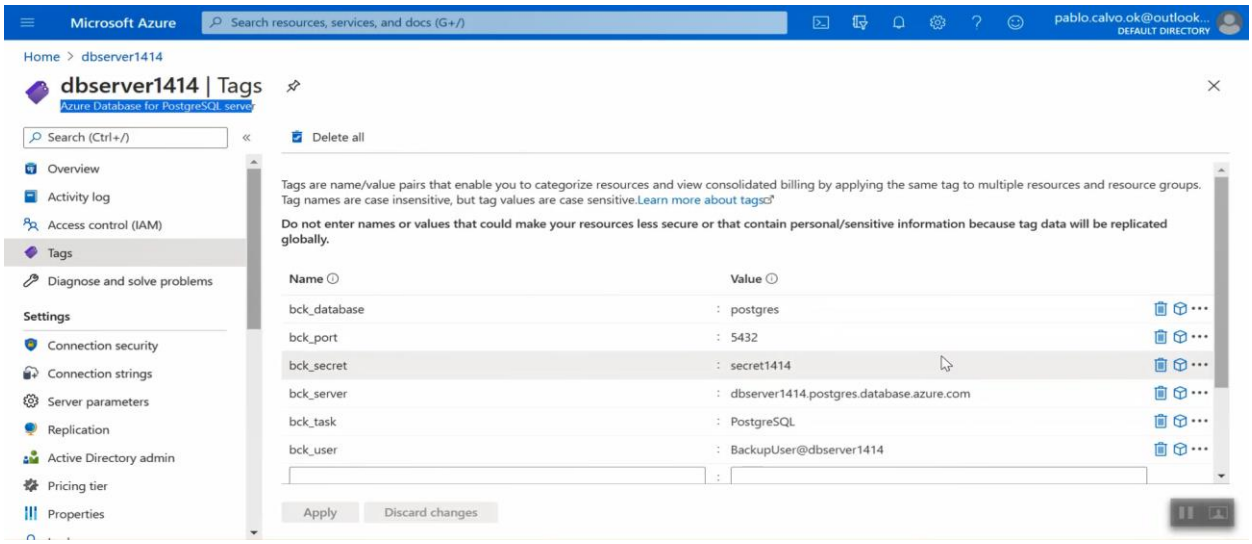


Figure 6 - Resource tagging

1. Connection secrets

We are using the vault mechanism to avoid password exposure. Vault allows separate responsibilities regarding the definition of access and use of the information.

Figure 7 shows the Key vault keyvault1414 that matches the keyVaultName key-value. keyvault1414 contains the secret1414 entry (Figure 8) that matches the tag key bck_secret configured in the PostgreSQL resource shown previously.

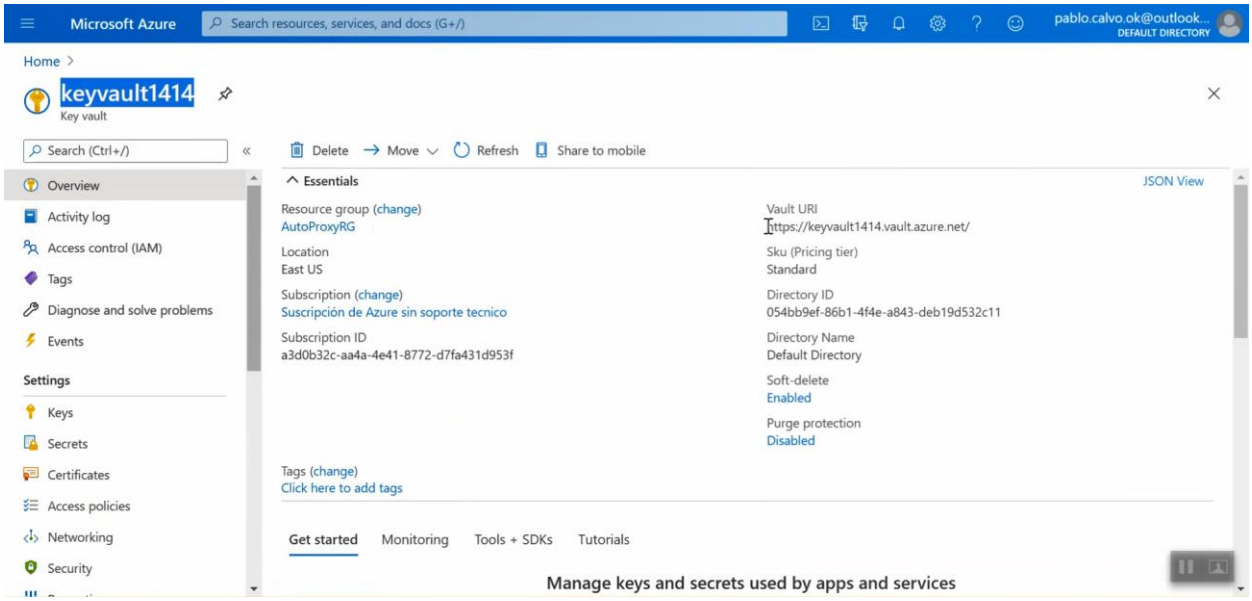


Figure 7 - Azure Key vault configuration overview

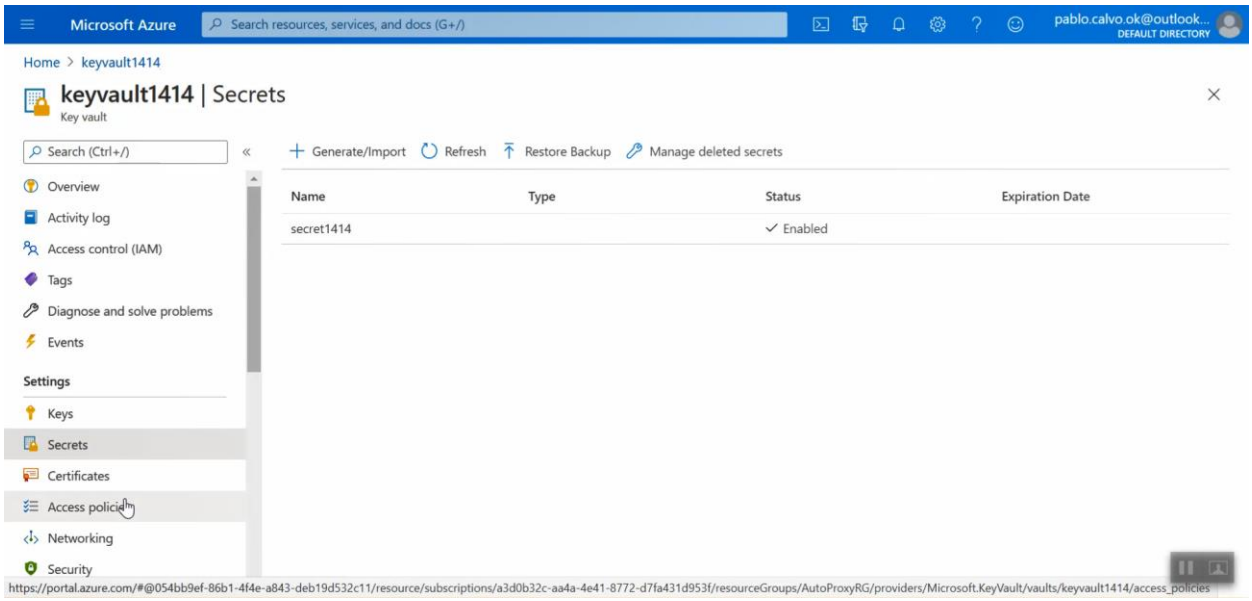


Figure 8 - Secret configuration

Example script to get security token[17]. The code shown below gets a token using Azure Instance Metadata Service (IMDS) endpoint via HTTP. The address (169.254.169.254) used is the managed identities for Azure resources endpoint for the Instance Metadata Service. After that token is recovered we consult the secret to bring us the required password.

```
KeyVault=`cat dps-setup.json  | jq -r '.keyVaultName'`
cat ${ConfigDir}/swoconfig | while read linea
```

[17] Based on original script provided by SoftwareOne

```
do
   set -a $linea " "
   if [ "${1::1}" != "#" ] ; then
      ERROR=0
            echo !!!!! Processing token from Keyvault !!!!!
            response=$(curl 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-
            01&resource=https%3A%2F%2Fvault.azure.net' -H Metadata:true -s)
      if [ ${response:2:5} == "error" ]; then
         echo " ERROR 001 Getting token from KeyVault "
         ERROR=1
         break
      fi
            access_token=$(echo $response | python3 -c 'import sys, json; print (json.load(sys.stdin)["access_token"])')
      echo !!!!! Processing value from Keyvault !!!!!
            response=$(curl https://${KeyVault}.vault.azure.net/secrets/$6?api-version=2016-10-01 -s -H "Authorization: Bearer
            ${access_token}")
        if [ ${response:2:5} == "error" ]; then
           echo " ERROR 002 Obtaining key value from KeyVault "
           ERROR=2
           break
        fi
            pass=$(echo $response | python3 -c 'import sys, json; print (json.load(sys.stdin)["value"])')

fi
done  < ${ConfigDir}/swoconfig
```

Backup control data and data flow

The Docker container has the following components installed inside:

- An Avamar client (we could use other software if necessary).
- A DDBoost File System client (we could configure data dumps to NFS if necessary to avoid DDBoostFS).
- An Azure client to discover Azure resources.
- A PostgreSQL client used to extract the data from the chosen database (or a MySQL client to back up MySQL database and so on).

This container is registered in Avamar as a client, enabling backup administrators to work with this like a “regular” client.

Figure 9 show that "dockerpg-01" client has been registered in the domain "/clients".

Figure 10 shows that “dockerpg-01” client has been included in a backup policy.

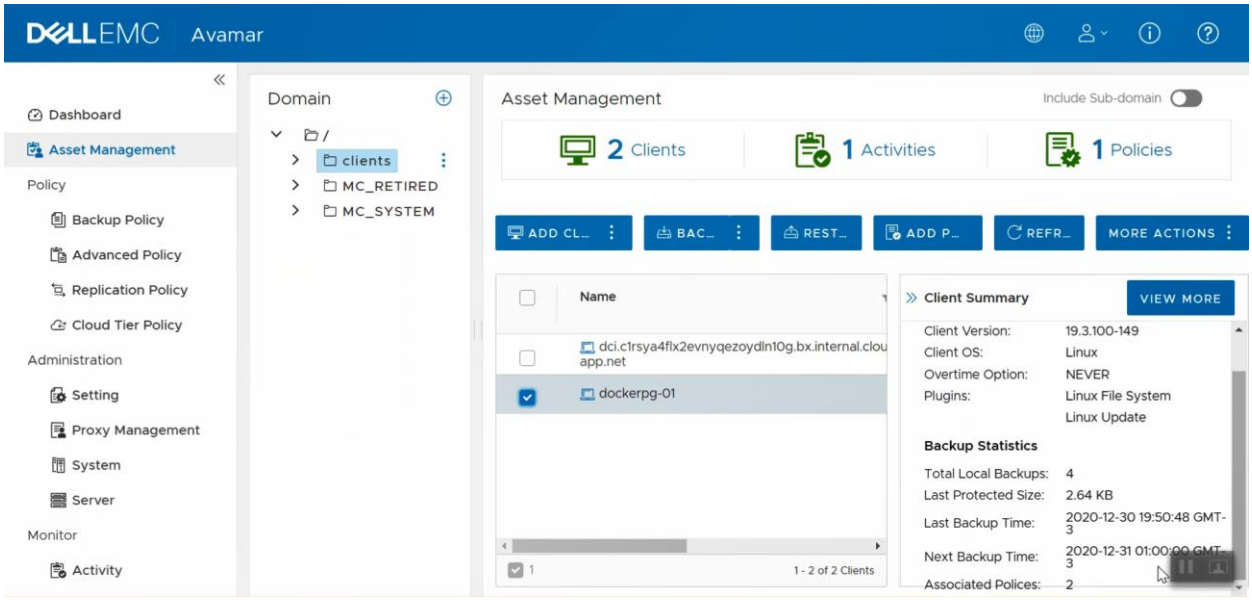


Figure 9 - Avamar container client configuration

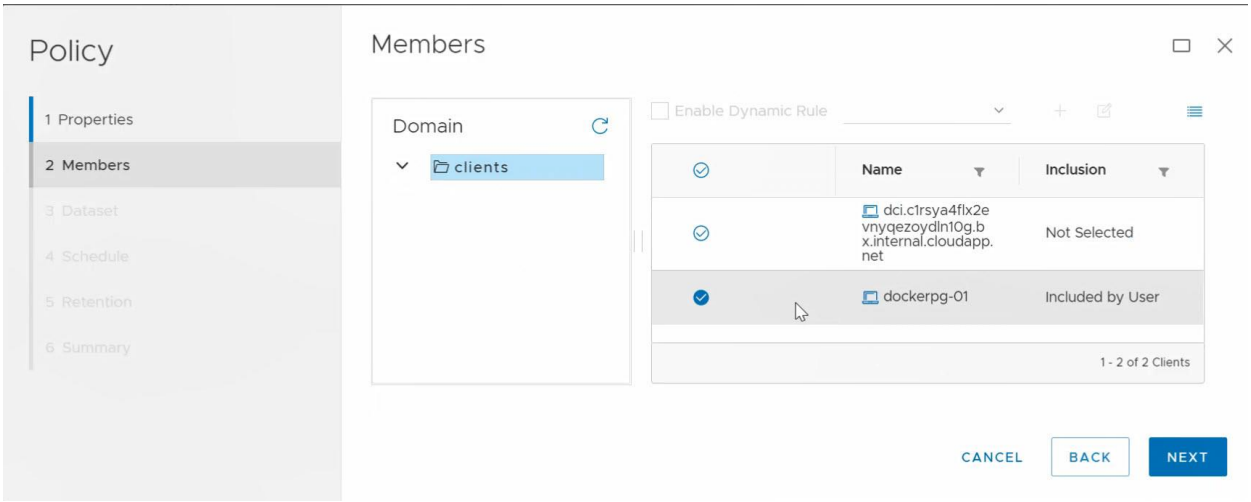


Figure 10 - Avamar container policy configuration

This part of the code shown below dumps the data contained in the PostgreSQL database to the destination indicated by the datadomain.RootBackupDir key of the JSON configuration file. Note that the access key is not exposed at any time

```
RootBackupDir=`cat dps-setup.json  | jq -r '.datadomain.RootBackupDir'`
ServiceBackupDir=${RootBackupDir}/$SERVICE_TYPE
BackupDir=${ServiceBackupDir}/backups
cat ${ConfigDir}/swoconfig | while read linea

do
set -a $linea " "
PGPASSWORD=${pass} pg_dump -Fc -v --host=$3 --username=$5 --dbname=$4 -f ${BackupDir}/${2}.dump
fi
done  < ${ConfigDir}/swoconfig
```

pg_dump is the PostgreSQL dump utility [18]
Figure 11 shows the configuration of the Storage Unit of the Data Domain used to store the data. This data will be stored in the mtree associated with this SU which differs from the Avamar mtree.

The Avamar backup task is executed to catalog this backup information. Due to the Data Domain global deduplication feature, data is not rewritten at the time of backup execution.

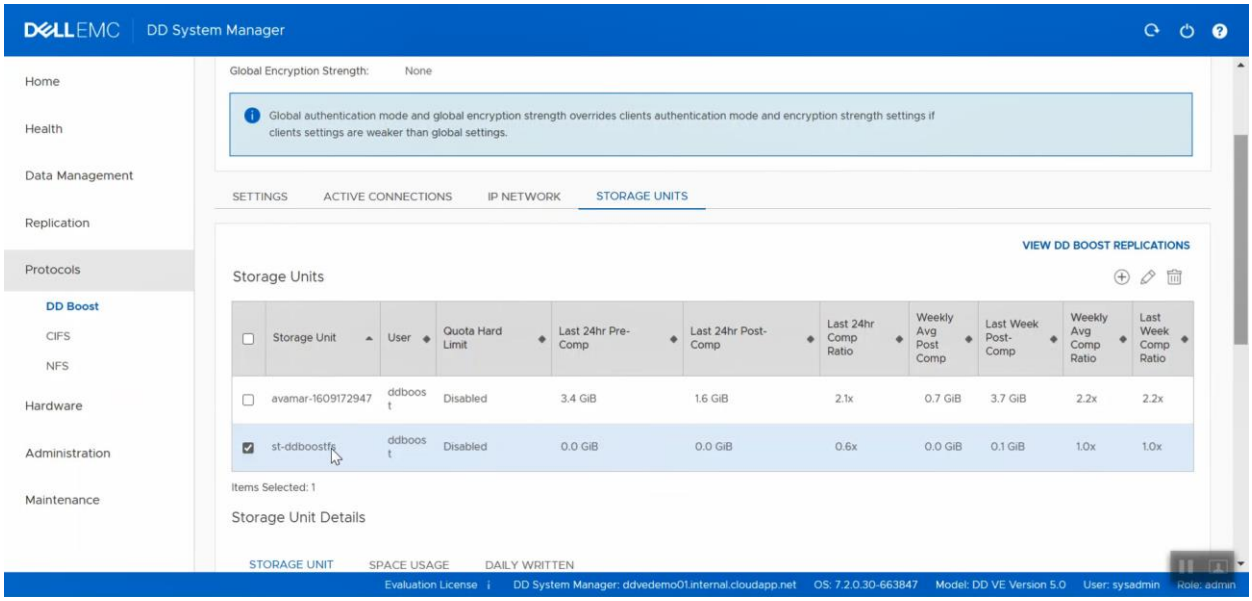


Figure 11 - Data Domain storage unit configuration

[18] pg_dump is a utility for backing up a PostgreSQL database (https://www.postgresql.org/docs/9.1/app-pgdump.html)

Backup catalog data

Figure 12 shows the configured path to which Avamar will go to read the dump files.

Figure 13 shows the user-configured script stored inside of the container. This script does:

- Automatic Discovery of tagged resources.
- Password recovery from the defined vault.
- Backup data dump.

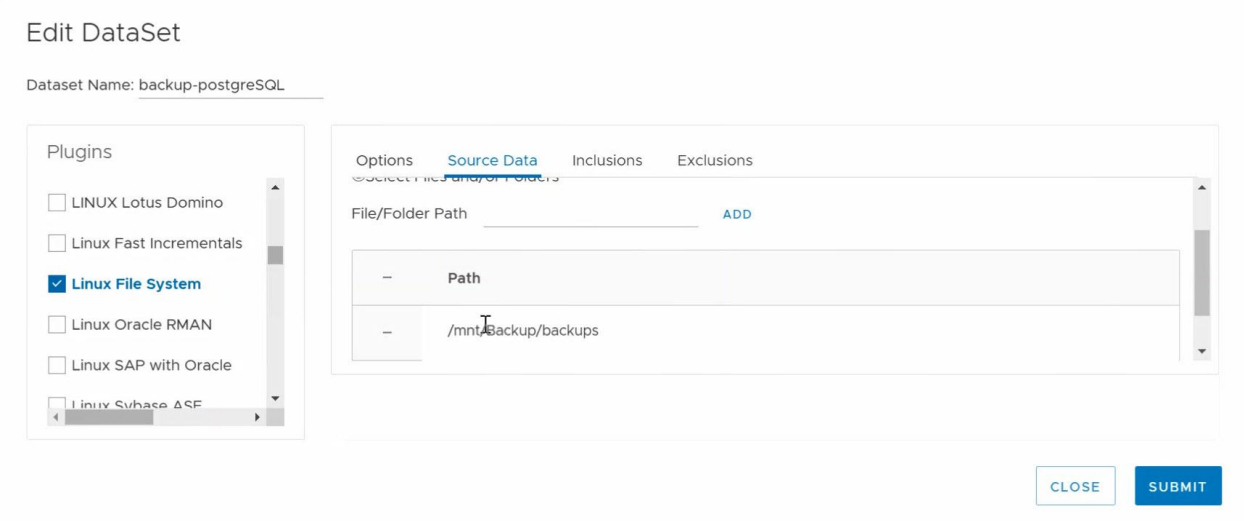


Figure 12 - Folder to backup

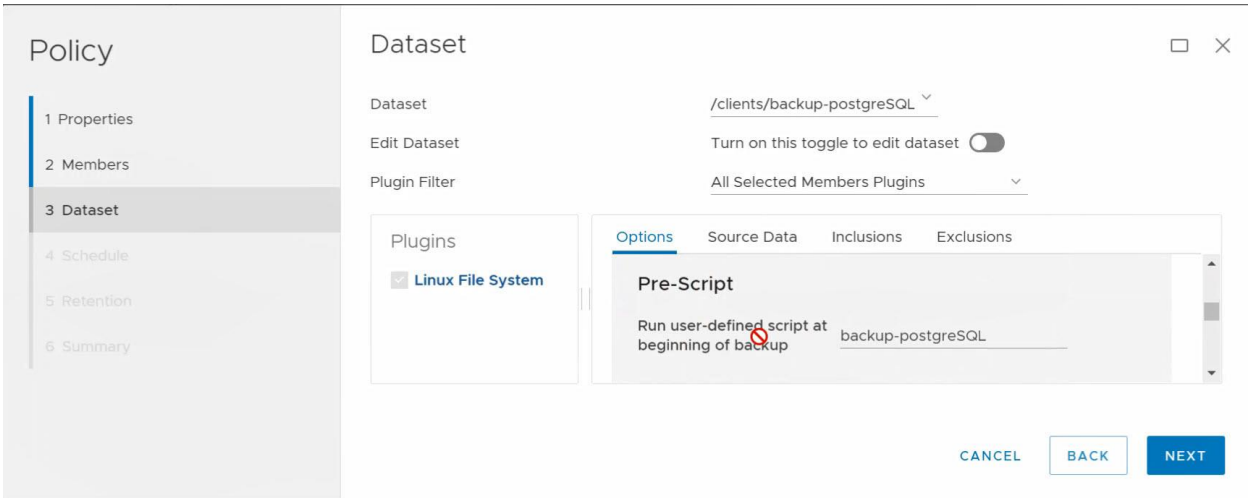


Figure 13 - User-defined script configuration

Additional Cloud Provider Scenarios

The architecture that has been explained in this article can be applied to different cloud providers, such as Google Cloud Platform and Amazón Web Services.

The main difference between the case depicted is the way to interact with the deployment of the VM and their target resources to be backed up.

The tools to access those resources are the AWS client and the Google Cloud Platform SDK. Resources from those platforms are compute, keyvaults and access to the API to read the resources to be protected.

In summary a container platform hosted on a Virtual Machine or a Container Platform like Kubernetes or Openshift can be used to configure the functionalities provided by this autodiscovery backup proxy.

Summary

The main features of this solution are the portability and flexibility of its configuration. As a containerized system, it can be deployed with DevOps methodologies.

The architecture can be connected to different sources of traditional backup providers, customizing and configuring the backup clients. It also provides the benefit of working with Data Domain systems, physical (on prem) or virtual, to get the best deduplication and performance.

This proxy only requires a configuration file and enables automated deployment in minutes since it can be integrated with any orchestration tool on the market. It can be integrated with virtually all current technologies that cloud providers offer as a service, including databases (MySQL, PostgreSQL, Cosmos DB) and blob storage, among others.

Like a small resume we can group the characteristics of our proxy into four groups that relate to integrations, infrastructure, deployment and costs and in each of these groups we will find the following differentiators:

INTEGRATION

1. Dell Technologies Data Protection Solutions Integration capability.
2. Backup & restore integration to data sources.
3. Any cloud dataset that can be discovered and accessed by Rest API.

INFRASTRUCTURE

4. Container and virtualized infrastructure.
5. Orchestrated on Kubernetes or similar.
6. Cloud platforms: Azure, AWS, GCP, DO.

DEPLOYMENT

7. One parameter file to config.
8. Automated deployment.
9. Replication between cloud providers.

COSTS

10. Low Total Cost of Ownership (TCO).

References

1. Video demo https://youtu.be/isaHN4K6Quk
2. Code repository https://github.com/uniqs-devops/bkp-proxy.git
3. Leveraging Avamar for File Level Backup of apps running on Kubernetes https://github.com/cn-dp/K8s-Avamar
4. Data Protection: Avamar, NetWorker, Data Domain, RecoverPoint, PowerProtect, CSM https://nsrd.info/blog/2019/03/05/proof-of-concept-docker-with-boostfs/
5. Azure managed identities https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/how-to-use-vm-token
6. Azure to AWS map: https://itnext.io/azure-to-aws-map-70d4c56f55a7
7. Google Cloud vs Azure Features Comparison: https://kinsta.com/blog/google-cloud-vs-azure/
8. Cloud vendor products
    a. Amazon AWS: https://aws.amazon.com/products/
    b. Google Cloud Platform (GCP): https://cloud.google.com/products/
    c. Microsoft Azure: https://azure.microsoft.com/services/