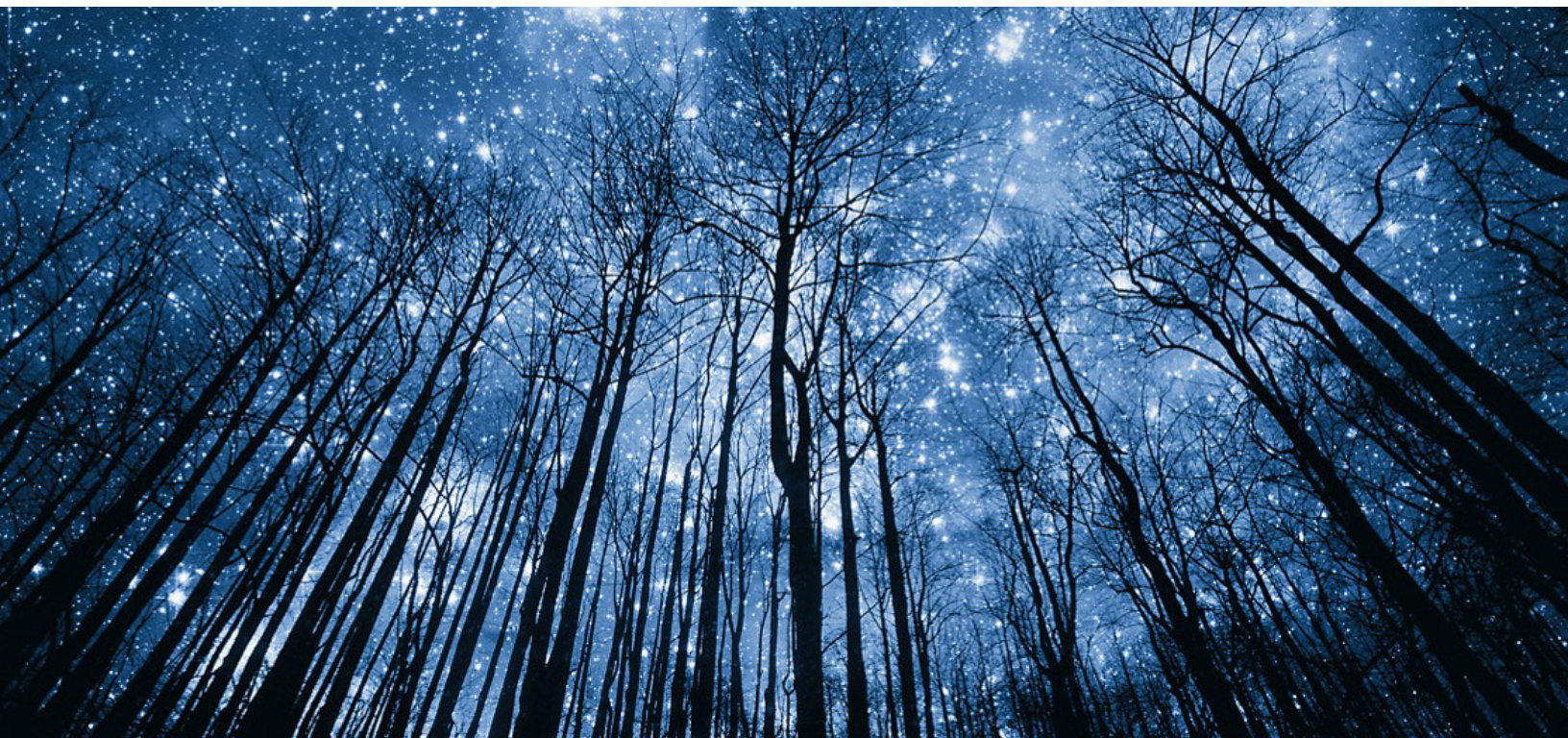


A GUIDE TO PROTECTING ANY DATA ON ANY CLOUD



Pablo Calvo

Services Provider
Dell Technologies

The Dell Technologies Proven Professional Certification program validates a wide range of skills and competencies across Dell's multiple technologies and products with both skill and outcome-based certifications.

Proven Professional exams cover concepts and principles which enable professionals working in or looking to begin a career in IT. With training and certifications aligned to the rapidly changing IT landscape, learners can take full advantage of the essential skills and knowledge required to drive better business performance and foster more productive teams.

Proven Professional certifications include skills and solutions such as:

- Data Protection
- Converged and Hyperconverged Infrastructure
- Cloud and Elastic Cloud
- Networking
- Security
- Servers
- Storage
- ...and so much more.

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.

Table of Contents

Executive Summary	4
Introduction	5
Objectives	5
Cloud Protection Needs.....	6
Cloud computing reference model	6
Users create data all the time	7
Let's Explore a Solution.....	8
Cloud login/get login credentials/cloud logout.....	8
Labeling/resource scanning/exhaustive enumeration	12
Traffic	18
Platform Design Guidelines	19
How to design a platform to include all application types.....	19
Components of a container	22
How to orchestrate the solution.....	25
Conclusion	41
Appendix.....	43

Executive Summary

What would you say if you managed to build a data protection platform that allowed you to adapt every day to the new requirements of the industry? Remember, the discipline of data protection does NOT influence the creation of new data processing technologies.

Data may be created by someone else, in another place, at another time, and without our knowledge.

Imagine if we were able to combine all the traditional data protection tools and practices that for years have been carried out successfully in on-premise installations, without resigning essential characteristics such as scalability, various restore techniques, the ability to replicate backup servers transparently, air gapping, and ransomware protection; if we were able to have a single backup record for any data generated in our organization regardless of where it is generated. What if we were free to choose one single backup software, and if we did not care that the cloud we opt for today is this hybrid, public, private or multiple, and did not forget to try to continue using the already mature deduplication algorithms at source and destination?; In short, if we did not resign the path we have traveled, but were able to adapt it to the new demands, we would have managed to protect our organization's data as our organization requires, in a more transparent manner.

We should not need to give up the freedom to store our copies of data in the place we have chosen and with the technology that best meets our needs. Why do we have to leave our documents guarded by the same provider that provides us with the cloud service? What would happen if that provider disappeared, had a severe service disruption, or if we decided, we didn't want to continue contracting it? Where could we manage at least one copy of our data, as we have been proclaiming for years? If we want to, but do not choose the best for our organizations, we have granted too much to third parties. They have the original data and all our copies.

This document will teach you how to build a universal backup platform. If you wish, you can develop your product to be capable of adapting to any future demand for data protection based on these guides; you'll also be able to create a "cloud-aware" layer that works as a front-end to your current backup product, which was built to protect on-premises assets.

We define a cloud-aware layer as the portion of computer code that allows you to send a copy of original data from any cloud service, from any cloud to the repository and backup software that we decide to use. This layer is highly dynamic and will be modified and adapted many times because the offer of new services currently imposes it. We do not control the number of new offers, and we have to adapt quickly to changes. Cloud providers set new standards, and collaborative programming speeds up the creation of new technologies. We must adapt as soon as possible to give our customers a good data protection solution.

The time for monolithic backup solutions based on in-house non-collaborative development teams is gone. There is no time to wait for a new backup product version every six months.

Introduction

Objectives

The main objective of this document is to show a flexible data protection solution platform that quickly adapts to changes in the industry that allow adding:

- The new applications (use cases) as the sector creates them.
- Possibility to adapt to new data extraction methodologies at cloud providers, such as using REST or https connections.
- New cloud providers.

The solution must integrate with the selected backup software solution and support different data repositories to maintain information based on schedulers and retention policies.

The backup application software must be able to be installed in a cloud environment with an infrastructure-as-a-service format.

It must be cheap, easy to deploy, and fault tolerant.

Cloud Protection Needs

Cloud computing reference model

Figure 1 represents the computational architecture model developed by NIST and identifies the leading players, their activities, and functions within the cloud-computing environment in a high-level diagram. This diagram defines the terms and concepts we will use in the following sections of the document.

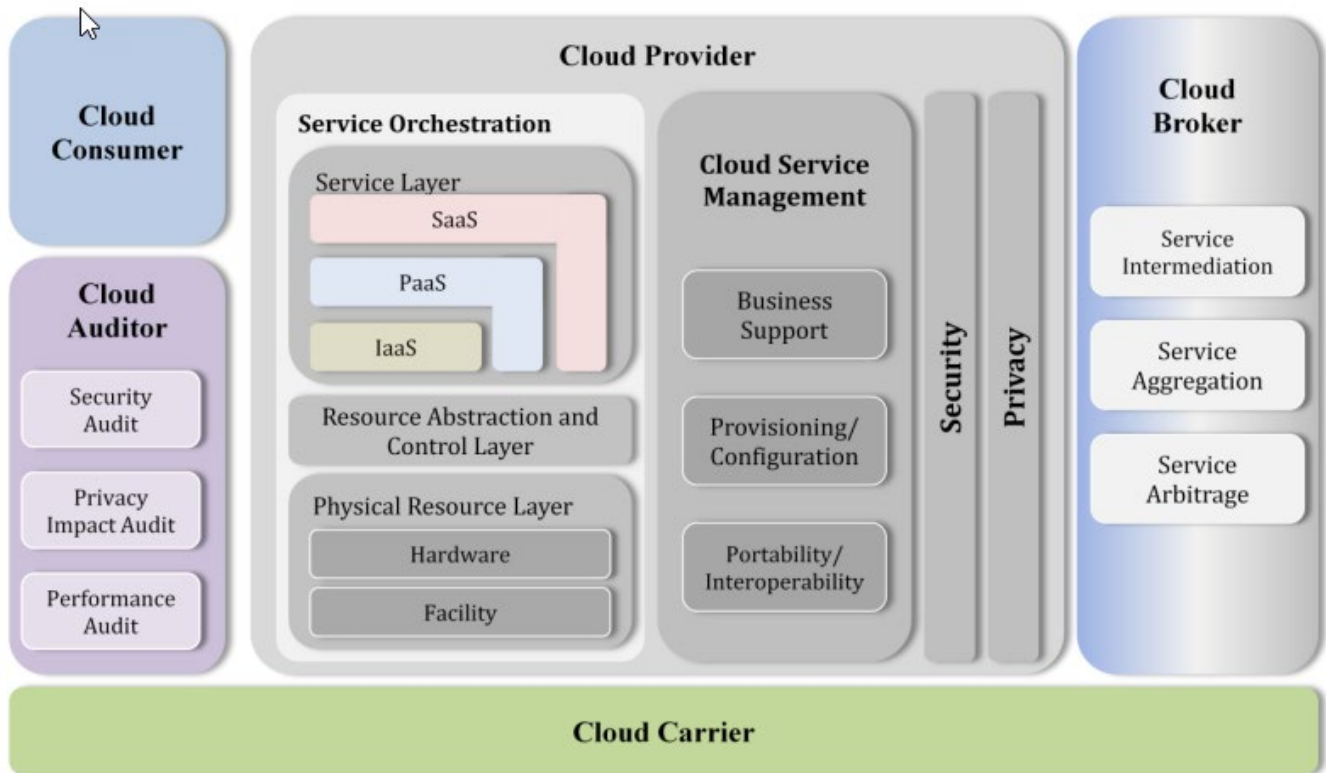


Figure 1

This diagram defines five main actors: the **consumer**, the **cloud service provider**, the **carrier**, the **auditor**, and the **broker**. Each interacts in some way with the resources available in the cloud.

Cloud Consumer: A consumer is a person or an organization that uses services provided by a cloud provider. The consumer searches for services from a catalog or marketplace and requires them from an initial configuration that is carried out. Services can be billed based on your consumption.

Cloud provider: The cloud provider is an organization that offers cloud services. If the service type is software-as-a-service (SaaS), then the cloud provider implements, configures, and modifies the operation of the software within its infrastructure. In these cases, the provider manages and controls the application and their entire infrastructure. In the platform-as-a-service (PaaS), the provider offers the venue and the software stack that runs over this platform.

The PaaS Provider also supports the development, deployment, and management by providing tools such as integrated development environments (IDEs), development versions of cloud software, and software development kits (SDKs). Management and deployment tools are provided too.

The last and most basic service is infrastructure (IaaS); in this case, the provider provides servers, networks, and storage for the consumer to run their applications.

Cloud Auditor: The auditor runs independent tests on cloud services to verify that they meet the consumer's specific standards. The goal of cloud audit is to increase the security level to avoid vulnerability of the assets.

Cloud Broker: The broker is an integrator of more than one service according to the consumer's requirements. Among the most common services, we find intermediation of services, aggregation of services, and arbitration.

Cloud Carrier: This last actor, the carrier, is the one that provides connectivity to cloud services, carrying out the task of transport between providers and consumers. It is essential to ensure that end-to-end traffic is secure to avoid potential information being sniffed by malicious third parties.

In the next section, we will see in more detail how these factors interrelate, how some produce data and others consume it, and why data protection continues to be of vital importance.

Users create data all the time

The previous model shows us different actors within the reference architecture of information processing in the cloud.

Each one performs various tasks or fulfills well-differentiated roles. Still, they all have one component in common: the data created by consumers represents their most precious value; the cloud provider protects the data generated, and the cloud carrier transports the info from the physical cloud to the user application interface.

Eventually, the auditor may require them for some analysis. We also have brokers that integrate these original services into more complex services, such as data governance services.

The data owner is the consumer, and no more; the other four actors provide services related to obtaining and exploiting data. However, it may be a desire of the consumer to have some independence from the cloud provider and the associated services, either by the broker or by the cloud provider itself in any of the previously mentioned modalities (SaaS, PaaS, or IaaS).

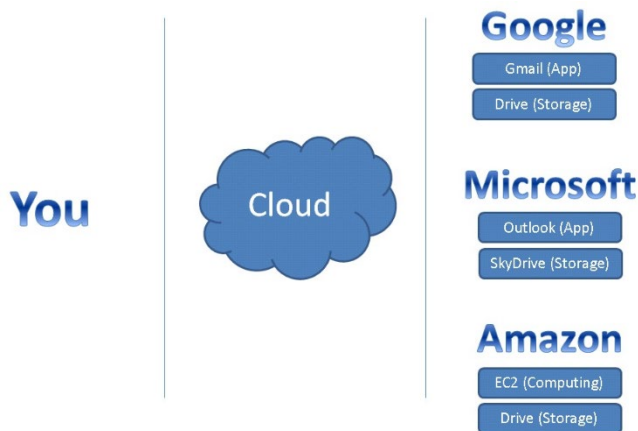
The previously described scheme needs to consider a fundamental aspect related to how to have a backup copy of the data generated or produced in the cloud. The ability to take backup in a unified manner has been lost so that the consumer continues to have a unified backup registry or database as in the on-premises forms.

The consumer does not have to be forced to delegate the protection of their data to the cloud provider(s) that accept more than one type of protection format, more than one repository, and more than one administration interface because they (the providers) do not have the capability of unifying the data protection process in one universal tool that understands the different technologies. Currently, clients need to have additional backup techniques, procedures, and technologies according to their chosen service and provider. We have gone back many years; we already had this resolved, and we already had a single backup platform for the entire organization.

In the next section, we will explore a solution that solves these problems and can provide the consumer with what they need: A universal cloud data protection solution capable of protecting any data with just a few configurations from their side.

Let's Explore a Solution.

Cloud login/get login credentials/cloud logout.



Cloud validation is the starting point for finding and allowing access to the resources we want to protect. ; The cloud access administrator must precisely define the necessary permissions to preserve and recover the information correctly.

The suggested login type is read-only or "backup" for data protection and write tasks or "restore" for data recovery tasks.

Validation methods include passing usernames and passwords, Service Principal Names, Managed

Identities, MFA, etc. CLI or API as access methods is the way to implement it.

The backup application to be designed must include all forms of validation under the security requirements of each installation, each cloud, and each customer.

Usually, all methods require access to a vault to obtain secret keys or certificates unless we use a managed identity system.

We will propose a set of JSON keys that allow the backup administrator to configure all the necessary parameters so that the backup solution can be validated in the cloud in the following example in Azure.

```
"azureLogin": {
  "resourceGroup": "",
  "tenantId": "",
  "ServicePrincipal": {
    "useServicePrincipal": "YES",
    "servicePrincipalClientId": "",
    "KeyVaultyes": { "loginKeyVaultName": "", "secretSPN": "" },
    "KeyVaultno": { "servicePrincipalClientSecret": "" }
  },
  "ManagedIdentity": {
    "useUserAssignedRManagedIdentity": "NO",
    "ManageIdentityName": "",
    "useSystemAssignedManagedIdentity": "NO"
  },
  "Credentials": {
    "useCredentials": "NO",
    "userName": "Pablo.Calvo@dell.com",
    "Password": ""
  },
  "subscription": {
    "changeDefaultsubscription": "NO",
    "subscriptionID": ""
  }
}
```

This JSON structure allows us to define if the login will use:

- A Service Principal Name ([ServicePrincipal.useServicePrincipal](#)).

Avoid the key "[KeyVaultno](#)" because it requires putting the SPN key in plain text in "[servicePrincipalClientSecret](#)". You can use this option only in test environments and with SPNs that differ from the production ones.

Use "[KeyVaultyes](#)" by filling the values "[loginKeyVaultName](#)" with the name of the key vault to use and "[secretSPN](#)" with the name of the secret that holds the SPN key.

The "[KeyVaultyes](#)" method is the best for production, pre-production environments, or those cloned, and you suspect may have sensitive access data.

- A Managed Identity ([ManagedIdentity.useUserAssignedManagedIdentity](#) or [ManagedIdentity.useSystemAssignedManagedIdentity](#)). A different and simple way to authenticate in the cloud is through managed identities which provide an automatically managed identity in the user directory and are thus used directly by applications when connecting to resources that support this type of authentication.
- Use "[useSystemAssignedManagedIdentity](#)" if you want to use a system-assigned identity on the resource (for example, a virtual machine) where you will run the backup solution. If you are going to use a user-managed entity instead, choose "[useUserAssignedManagedIdentity](#)" and fill in the value in "[ManageIdentityName](#)"

or

- A credentials login-based ([Credentials.useCredentials](#)). Use this option in basic test environments; the account does not have to require MFA.

For any of the outlined methods, it must be defined with the following parameters like tenant, subscription, and the default resource group.

Only a tiny piece of computer code is necessary to manage the different authorization mechanisms allowed by parsing the JSON file.

A logic called "**IF IS / TRY VALIDATION / THEN EXIT**" is suggested to validate the first "YES," If it is satisfactory, do not consider the other methods. The evaluation logic of a process configured with a JSON key value "NO" is "**IF NO/ NEXT METHOD**".

Let's see the basic programming logic in a simple-to-understand way; details are available on the GitHub repository reported at the end of the document.

```
if [ "Use Service Principal" = "YES" ]; then
    if [ "Using Service Principal and Key Vault" ]; then
        "get access token"
        if [ "Using Service Principal and Key Vault Secret" ]; then
            "get secret value"
                # Key from Key Vault Secret
                do splogin; return
            else [ "Using Service Principal without Key Vault" ]; then
                # Key from json file
                do splogin; return
            fi
        fi
    fi
fi
if [ "User Assigned Manage Identity Login" = "YES" ]; then
    echo
        do uallogin; return
else
    if [ "System Assigned Manage Identity Login" = "YES" ]; then
        do sallogin; return
    fi
fi
if [ "User Credential Login" = "YES" ]; then
    do credlogin
fi
```

Labeling/resource scanning/exhaustive enumeration



Finding resources within the cloud is the point. These resources are dynamically turned on and off, provisioned, and decommissioned without the backup application and administrator's control.

Then the application must be able to know all the resources that need to be protected from external labels that the DevOps administrator creates unattended.

An example tag could be:

If you cannot recognize resources based on labeling, we can opt for auto-discovery by searching within the subscription or context accessed after validation. This scanning method is also beneficial since it will allow us to understand all those resources that have been provisioned, regardless of whether they have labels configured or not.

The exhaustive enumeration is the third way of recognizing resources to be safeguarded, which is helpful for small configurations or non-productive environments.

Let's summarize the methods.

1. Labeling
2. Resource scanning
3. Exhaustive enumeration

Labeling method

Tagging is the lightest method of implementing resource lookup since the tag is just a property configured with cloud management commands. We can say many things with labels, among them if a resource should be protected and even configure more advanced features such as backup port, backup user, the vault secret where the credential is stored, and so on.

Let's see some JSON keys to handle labels and pass info to our backup application.

```
"backupTags": [  
  {  
    "type": "markedForBackup",  
    "value": "requireBackup"  
  },  
  {  
    "type": "user",  
    "value": "bck_user"  
  },  
  {  
    "type": "port",  
    "value": "bck_port"  
  },  
  {  
    "type": "database",  
    "value": "bck_database"  
  },  
  {  
    "type": "task",  
    "value": "bck_task"  
  },  
  {  
    "type": "secret",  
    "value": "bck_secret"  
  }  
],
```

For example, the type **"markedForBackup"** indicates that this customer has decided to use the **"requireBackup"** tag to announce whether their cloud resources require backup. In the same way, **"type": "port"** says that the **bck_port** tag informs the port (if needed) to use to protect the resource.

The other values mean:

- **user**: JSON key indicating that the `bck_user` tag informs the backup user.
- **database**: JSON key indicating that the `bck_database` tag informs the database that it requires backup and accepts "*" for all the resource databases.
- **task**: JSON key indicating that the `bck_task` tag informs the name you want to give to the backup job.

According to this setup, only a tiny piece of computer code is necessary to parse the JSON keys and look for tags configured on the cloud resources. We can consider the key "`type`": "`markedForBackup`" as mandatory; if it exists with a value other than space, it means that the resource discovery method is the "labeling".

The following computer code shows how to parse a JSON file with the above format and store the values of its keys in variables; once the variables have their correct values; it is straightforward to program programming logic.

```

USER_TAG=`jq '.backupTags[] | select(.type=="user")|.value' \
  $ConfigDir/dps-setup.json | sed 's//g'`
PORT_TAG=`jq '.backupTags[] | select(.type=="port")|.value' \
  $ConfigDir/dps-setup.json | sed 's//g'`
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'`
MARKED_FOR_BACKUP=`jq '.backupTags[] | \
  select(.type=="markedForBackup")|.value' \
  $ConfigDir/dps-setup.json | sed 's//g'`

tags=$(



```

Resource scanning method

Perhaps many customers do not use tagging to manage their cloud resources. For this case, we can use another method to list the resources we want to protect, and we will call it "**resource scanning**."

Like the tagging method listed above, we can govern the configuration of the resource scan with a few JSON keys; let's see the proposal:

```

"cloudResources": {
  "resourceType": "Microsoft.Sql/servers/databases",
  "useAutoDiscover": "YES"
},

```

Follow some scan commands to understand better the resource discovery based on resource scanning.

Microsoft Azure example:


```
az resource list --resource-type "Microsoft.Sql/servers/databases" \  
--resource-group PaaSBackup \  
-o table | tail -n +3 | awk {'print $1'} > ${ConfigDir}/resources
```

Here we scan all resource types `"Microsoft.Sql/servers/databases"` of resource group `PaaSBackup`, and the output is stored in `${ConfigDir}/resources` file. A backup loop in the same script may process this file.

Exhaustive enumeration

In some circumstances, we can avoid using tagging or resource scanning methods. For example, we have identified a resource or a couple that interests us and none else because they represent a greater criticality or because the amount of information to back up is enormous or that requires a scheduler that places them in a separate backup window, etc. we can use the `"type": "resource_list"` key on the `fixValues` section to enumerate the cloud resources for which backup is required; `resource_list` is a comma-separated list.

```
"fixValues": [  
  {  
    "type": "user",  
    "value": "xxxxxxx"  
  },  
  {  
    "type": "port",  
    "value": "xxxxxxx"  
  },  
  {  
    "type": "resource_list",  
    "value": "*"  
  },  
  {  
    "type": "database",  
    "value": ""  
  },  
  {  
    "type": "task",  
    "value": ""  
  },  
  {  
    "type": "secret",  
    "value": "xxxxxxxxx"  
  }  
]
```

The following computer code shows how to parse a JSON file with the above format and store the values of its keys in variables; once the variables have their correct values; it is straightforward to program programming logic.

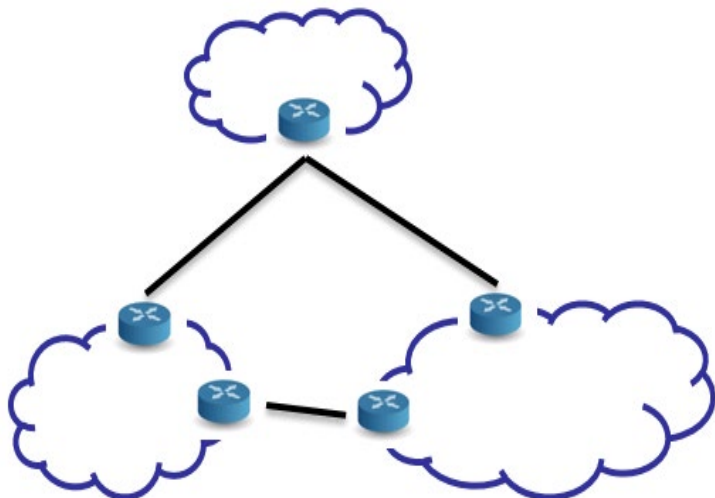
```
USER_FIX=`jq '.fixValues[] | \
  select(.type=="user")|.value' \${ConfigDir}/dps-setup.json | sed 's/"//g'`
PORT_FIX=`jq '.fixValues[] | \
  select(.type=="port")|.value' \${ConfigDir}/dps-setup.json | sed 's/"//g'`
RESOURCELIST_FIX=`jq '.fixValues[] | \
  select(.type=="resource_list")|.value' \${ConfigDir}/dps-setup.json | sed 's/"//g'`
SERVER_FIX=`jq '.fixValues[] | \
  select(.type=="server")|.value' \${ConfigDir}/dps-setup.json | sed 's/"//g'`
DATABASE_FIX=`jq '.fixValues[] | \
  select(.type=="database")|.value' \${ConfigDir}/dps-setup.json | sed 's/"//g'`
TASK_FIX=`jq '.fixValues[] | \
  select(.type=="task")|.value' \${ConfigDir}/dps-setup.json | sed 's/"//g'`
SECRET_FIX=`jq '.fixValues[] | \
  select(.type=="secret")|.value' \${ConfigDir}/dps-setup.json | sed 's/"//g'`
```

Traffic

End-to-end security is an essential aspect to consider, and it is mandatory to support encryption.

A helpful solution needs to support endpoints and network proxies.

To reduce costs, the traffic between cloud resources and the backup application must be deduplicated at the source.



```
"cloudConnection": {  
  "proxy": {  
    "useProxy": "YES",  
    "proxyHttpName": "http://proxy.com:443",  
    "proxyHttpsName": "https://proxy.com:443",  
    "noProxy": ""  
  },  
  "certs": {  
    "useCerts": "YES",  
    "certFile": "Root_CA.cer"  
  },  
  "EndPoints": {  
    "useEndPoints": "NO",  
    "EndPoint": "xxx.xxx.xx.xx"  
  }  
},
```

Use previous techniques to be able to manipulate the values of the JSON.

Platform Design Guidelines

How to design a platform to include all application types

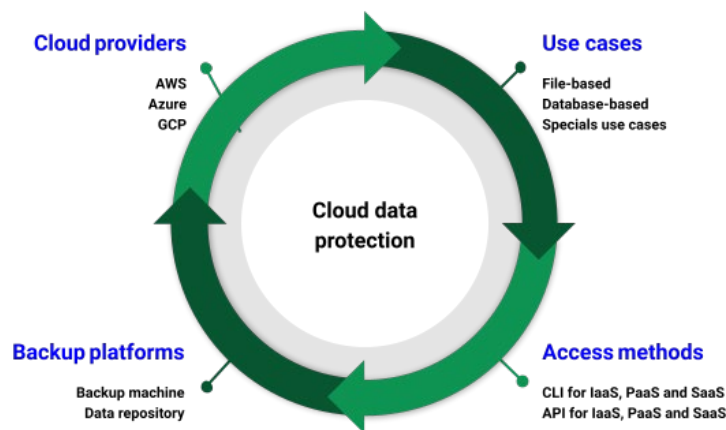
If we want to achieve adequate data protection in the cloud, it is necessary to consider four different aspects.

The first of these is the **cloud provider**, which is the one who provides the infrastructure where the data is hosted.

We must realize that there may be more than one cloud provider in a complex customer installation; each of these cloud providers markets distinct types of applications in three modalities: software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS).

The platform must allow adding new **cloud providers** to the backup platform according to the client's wishes with all the procedures validation and discovery of resources of this provider.

There may be more than one cloud provider in a complex customer installation; Each of these cloud providers markets different types of applications in three modalities: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).



These providers or third-party manufacturers provide applications; we will call these applications **use cases**. A use case is a business application chosen by an organization to process its data. The division of cases is into three groups according to the way of accessing the data:

- Applications based on files

File-based applications generate information that needs to be stored in flat files and, consequently, can be accessed more efficiently; it does not require a database engine or other

complex program to read the generated data.

Within this group, we will name Azure Storage Account, Azure ADLS, Google Cloud Storage, S3, MinIO, NFS (network file system), and CIFS (Common Internet File System), among others.

- Database applications

These represent data grouped in database structures (SQL or non-SQL). Examples within this group of use cases include MariaDB, PostgreSQL, Atlas, CosmoDB, Azure SQL, DynamoDB, MySQL, Percona, etc.

- In the third group, we find special cases that simply cannot be grouped in the previous two. Usually, in this group, we see data governance applications, cognitive services, and whose configurations are protected in XML, JSON, YAML, etc. format.

A robust data protection system must learn all the new applications that a customer decides to move to the cloud of their choice (**cloud providers**). Only the option remains to understand automatically or semi-automatically what further data is generated by which application they must protect. So, from now on, we must know where and how the information is stored and how it is generated so that we can take a copy and restore it, respecting the complete business logic of the application.

The way to access, the data will give rise to our **access methods**.

All this data must be stored within a **backup platform** that includes a backup machine and a repository where the copies are stored. In this way, we will have a single storage with all the backup records of our client, and there may be a platform for each cloud provider to reduce costs related to backup traffic between different cloud providers. The backup machine and the repository must have deduplication features to minimize storage and traffic costs between components.

Once we have the chosen backup solution, we will take advantage of some of the features that have been developed for a long time by the industry in this software product type; among them, we can name data compression and deduplication.

A more detailed list of features that a "traditional" backup solution should include are the following:

- Air-gapping to protect against ransomware attacks.
- Centralized Backup that supports a large number of data and compute environments.
- Integrated Deduplication to eliminate duplicate copies of repeating data
- The security includes encryption, lockbox access control, and a robust user authentication method.

- Performance that ensures fast copies.
- Replication to protect against disasters.

Components of a container

Each deployment will be done in a Docker container. Docker is a software platform that allows you to build, test, and deploy applications quickly. Docker packages software into standardized units called containers that include everything needed for the software to run, including libraries, system tools, code, and runtime. With Docker, you can quickly deploy and scale applications in any environment, confident that your code will run. Containers are an extremely cheap way to deploy code in the cloud and can be hosted on virtual machines, Kubernetes clusters, and more.

Let's see the details of the software components that will be part of a container.

A Cloud client that interacts with the **cloud provider**.

These are the native clients of each cloud. Among their functions, we will find validation by passing credentials, searching for resources, searching for secrets and keys, and asking questions to learn tags.

A Backup client to interact with the **backup platform**.

These are the clients of each one of the backup platforms that will initiate the backup tasks by sending the data to the platform. This way, all the configured properties will be inherited, such as scheduling of the arranged time, data expiration, replication of the backup record to the contingency platform, etc.

A DDboost client to interact with the **backup platform**.

In this writing, we have only included Data Domain as the data repository platform; in that case, we must include the DDBoost client in the client container. With the DDBoost client, a Data Domain Storage Unit will be mounted in the container to directly save the data from the dumps and exports of the databases. This way, we avoid requiring additional blob storage from platforms.

If the backup system is also connected to the Data Domain, the scheduled backup job will not re-duplicate the data due to the Data Domain global deduplication feature.

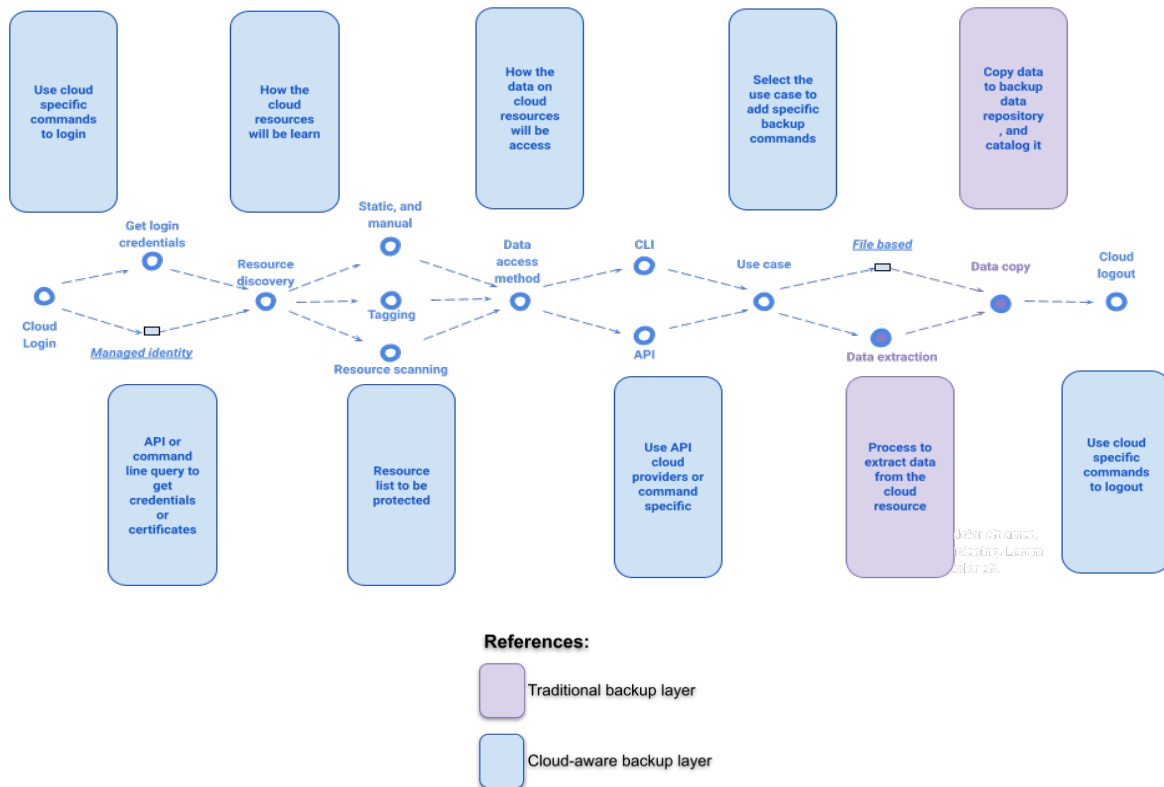
Use case-specific software (, PostgreSQL client, MySQL Client, FUSE client, and so on) to

interact with each **use case** using an **access method**.

These are clients provided by each manufacturer and are usually for products developed in an on-premises solution and then ported to the cloud. It is the portion of software installed inside each container, and usually, a username and password must be provided for the execution of the data extraction command.

What the container does

The following is the process flow that runs in the container: the complete backup logic that includes the "cloud-aware" layer and its communication with the "traditional" layer.



The container performs validation in the cloud through a **cloud login**, usually requiring **credentials** recovered via API.

Once the container has been validated in the cloud, it must learn or understand what resources (**resource discovery**) associated with that tenant and subscription need to be protected. This can be done in two ways, either from a **static and manual** configuration or through a self-learning method through the scanner of all the resources that are published (**resource scanning**). Additionally, this group could be reduced to those resources that have been labeled in a way that tells us that they require backup.

Knowing what resources, we are going to protect, we must choose the **access method** for data extraction. This method differs depending on the application and the different versions released by the manufacturers since the access methods are constantly changing, creating new ones that arise from migrating from a command-based (**CLI**) interface to an **API**-based one.

From the point of view of data access for protection (not to consume it as an application) we are going to classify them into two types:

- Based on commands: The way to access is based on the technology that the application manufacturer initially provided. Usually, these are the cases in most of the applications that were ported to the cloud. Some examples of command-based backups are mysqldump, pg_dump, SQL package, etc.
- Based on APIs: The other way to access data is to generate a copy based on the API application program interface. This method is usually provided for applications by cloud providers so that we can interact with a thinner protocol; for example, Azure Purview API, and Azure Cognitive Services APIs.

In the case of a database application, we must extract the data (**data extraction**) by some method and use intermediate storage to write the data. We suggest using a repository in the same tenant for the backup product. If the repository has the facility of global duplication, the data will be already stored, and it will only be a task of creating new pointers to the data.

If the use case is file-based, intermediate storage is not required since the data will be read directly from the data source.

Then, in both types of use cases, we execute a backup routine (data copy) with the traditional backup product we have chosen and catalog the data to manage its age, expiration, etc.

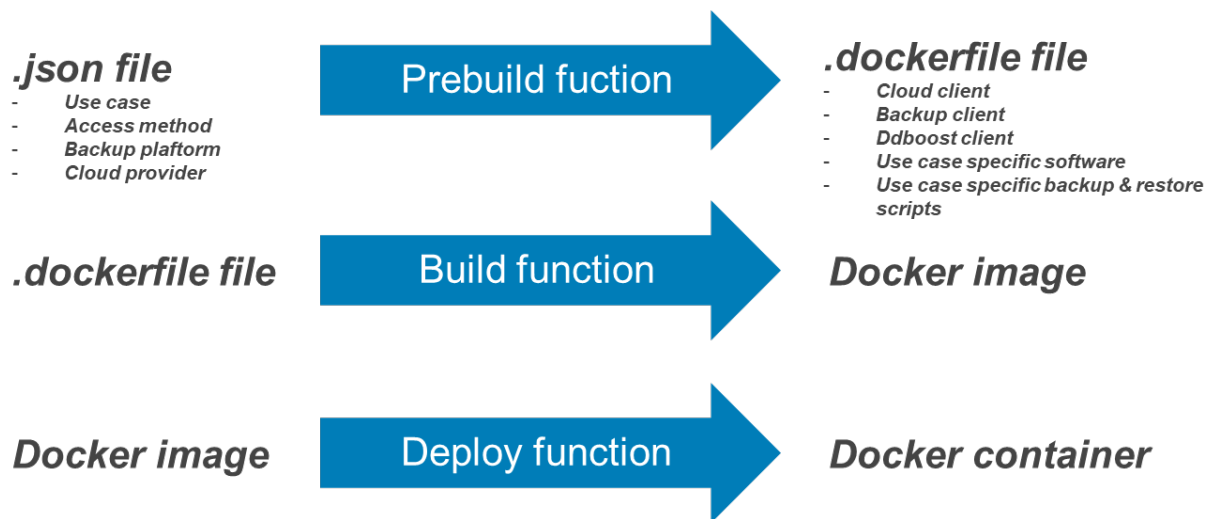
From these use cases, we will consider “**basic**,” more complex use cases we will call “**integrated**” because they will be a composition of more than one case; typically, you can make an application plus a database. The objective is to achieve, from an integrated use case, to program a single backup policy that generates a single business view of the data.

How to orchestrate the solution

All the logic that makes up the use case will be implemented within a container in this way, and thus, the container must be able to validate itself in the cloud, learn what resources it must protect, and choose the data extraction method to send them to the repository later. It must copy data so that once there, they are cataloged by the backup application.

The process of creating a container for a use case will start from a single JSON-type configuration file, and through a “pre-built” function, a docker-type file is obtained with all the components to include in the

image; a "Build" function will create the desired image to be deployed as a container by the "Deploy" function.



The "prebuild" function should do the following to complete the **cloud client** section of the new Docker file:

```
if [ cloud provider is Google Cloud ]
    then add the "google cloud cli" to install Google Cloud CLI software
else
    if [ cloud provider is Azure ]
        then add the "az cli" to install install Azure CLI fuse software
    else
        if [ cloud provider is AWS ]
            then add the "AWS cli" to install install Amazon CLI fuse software
        fi
```

After adding the cloud client, we must include the backup client and the DDBoost client with a programming logic similar to the following:

```

if [ backup client is Avamar ]
    then add the "avamar client" to install Avamar Client software
else
    if [ backup client is Networker ]
        then add the "Networker client" to install Networker client software
    fi
if [ data repository is Data Domain ]
    then add the "DDBoost client" to install DDBoost client software
fi

```

This may be the code to fill the **use case specific software** and **use case-specific backup & restore** lines.

```

if [ use case specific software is "minio" ]
then
    add the "rclone" line to install rclone software
    add the "mc" line to install MinIO client
    add the "backup-minio.sh" line to install MinIO backup script
    add the "restore-minio.sh" line to install MinIO restore script
else
    if [ use case specific software is "postgresql" ]
        then
            add the "postgresql" line to install PostgreSQL client
            add the "backup-postgresql.sh" line to install PostgreSQL backup script
            add the "restore-postgresql.sh" line to install PostgreSQL restore script
        else
            if [ use case specific software is "purview" ]
                then
                    add the "purviewcli" line to install Purview API interface
                    add the "backup-purview.sh" line to install PostgreSQL backup script
                    add the "restore-purview.sh" line to install PostgreSQL restore script
                else
                    if [ use case specific software is "cloudstorage" ]
                        then
                            add the "Cloud Storage FUSE" line to install Google Cloud Storage FUSE interface
                            add the "prebck-storage.sh" line to install PostgreSQL backup script
                            add the "postbck-storage.sh" line to install PostgreSQL restore script
                        else
                            if [ use case specific software is "keyvault" ]
                                then
                                    add the "backup-keyvault.sh" line to install PostgreSQL backup script
                                    add the "restore-keyvault.sh" line to install PostgreSQL restore script
                                fi
                            fi
                        fi
                    fi
                fi
            fi
        fi
    fi
fi

```

This way, we have a Docker file with all the desired software components; It is recommended that the base image be Alma Linux, but other flavors of Linux can be used without significant drawbacks.

We will develop three examples so the reader can see how creating the "cloud-aware" layer works.

Example 1: ***"We want to deploy a container to backup / restore Google Cloud Storage files and send/get the copy to/from Avamar."***

Docker file after prebuild function:

```

#!/bin/sh
# Get last fuse driver version from
#   https://github.com/GoogleCloudPlatform/gcsfuse/releases
FROM almalinux:latest
# Copy google-cloud-cli CLI
COPY packages/DockerEmbebed/blobstorage/google-cloud-sdk.repo \
  /etc/yum.repos.d/google-cloud-sdk.repo
# Install google-cloud-cli CLI
RUN yum install -y google-cloud-cli
# Copy avamar Client to /tmp for installation
COPY packages/DockerEmbebed/avamar/19.4/AvamarClient-linux-sles11-x86_64-19.4.*.rpm \
  /tmp
# Install avamar client usen RPM as Install Guide procedure
RUN rpm -ivh --relocate /usr/local/avamar=/dockerclient \
  /tmp/AvamarClient-linux-sles11-x86_64-19.4.*.rpm
# Copy Cloud Storage FUSE repo install package
COPY packages/DockerEmbebed/blobstorage/gcsfuse-*-*x86_64.rpm /tmp/
# Install Cloud Storage FUSE client
RUN yum install -y /tmp/gcsfuse-*-*x86_64.rpm
# Copy Syslog
COPY packages/DockerEmbebed/blobstorage/rsyslog-*x86_64.rpm /tmp/
# Install Syslog
RUN yum install -y /tmp/rsyslog-*x86_64.rpm
# json file
COPY dps-setup.json /dockerclient
# Copy includes, login and header
COPY src/avamar/azure/CLI/sources/*.sh /dockerclient/etc/scripts/sources/
# Copy backup script
COPY src/avamar/azure/CLI/prebck-storage.sh.sh /dockerclient/etc/scripts/prebck-storage.sh
COPY src/avamar/azure/CLI/preres-storage.sh.sh /dockerclient/etc/scripts/preres-storage.sh
COPY src/avamar/azure/CLI/post-mount.tmp /dockerclient/etc/scripts/post-mount.sh
RUN chmod 755 /dockerclient/etc/scripts/*.sh
# Housekeeping
RUN rm -rf /tmp/*

```

We can conclude the following after reading the Docker file generated by the pre-build function:

- The **cloud provider** is Google because this file installs google-cloud-cli.
- The **use case** is Google Cloud Storage because this file installs gcsfuse.

- The **backup platform** is Avamar because this file installs the Avamar-Client rpm file.
- We only know the **access method** once we review the backup or restore script that we have included in the container; in this case, we will see the backup script called **prebck-storage.sh**.

prebck-storage.sh script:

```
#!/bin/bash
version="1.0.0"
function mountgooglecloud {
#  echo "*** MOUNTING Google Cloud *****"
echo echoing $1 ${BackupDir}
gcsfuse --implicit-dirs $1 ${BackupDir}/$1
if [ $? != "0" ]
    then
        echo "`date +%Y%m%d.%T` Unable to mount backup through gcsfuse"
        exit
    fi
}
if [ $AUTODISCOVER = "YES" ]; then
    echo
    echo "*** SEARCHING All cloud resources ***"
    echo
    gcloud alpha storage ls --json | jq '.[].metadata.acl[].bucket' \
        | sort -u | sed 's"/"/g' > ${ConfigDir}/resources
else
    echo "*** LISTING cloud resources ***"
    echo
    echo $RESOURCELIST_FIX |fmt -1 |sed 's/,//g' > ${ConfigDir}/resources
fi
cat ${ConfigDir}/resources | while read linea
do
    set -a $linea " "
    if [ "${1::1}" != "#" ] ; then
        if [ ! -d ${BackupDir}/${linea} ];then mkdir ${BackupDir}/${linea}; fi
        echo mounting bucket: $linea into ${BackupDir}/${linea}
        mountgooglecloud $linea
    fi
done
```

... then:

- The **access method** is CLI because we use a command to access the data.

Example 2: *“We want to deploy a container to backup / restore Azure Key Vault secrets, download/upload an encrypted file on/from Data Domain and send /get the copy to/from Avamar.”*

Docker file after prebuild function:

```
#!/bin/sh
FROM almalinux:latest
# Copy AZ CLI client package
COPY packages/DockerEmbebed/azcli/azure-cli-*.x86_64.rpm /tmp
# Install AZ CLI
RUN yum install -y /tmp/azure-cli-*.x86_64.rpm
# Copy avamar Client to /tmp for installation
COPY packages/DockerEmbebed/avamar/19.4/AvamarClient-linux-sles11-x86_64-19.4.*.rpm \
 /tmp
# Install avamar client usen RPM as Install Guide procedure
RUN rpm -ivh --relocate /usr/local/avamar=/dockerclient \
 /tmp/AvamarClient-linux-sles11-x86_64-19.4.*.rpm
# Copy DDBoostFS
COPY packages/DockerEmbebed/ddboostfs/DDBoostFS*.rpm /tmp
# Install DDBoostFS
RUN yum install -y /tmp/DDBoostFS*.rpm && rm -rf /var/cache/yum
# json file
COPY dps-setup.json /dockerclient
# Copy includes, login and header
COPY src/avamar/azure/CLI/sources/*.sh /dockerclient/etc/scripts/sources/
# Copy backup script
COPY src/avamar/azure/CLI/backup-keyvault.sh /dockerclient/etc/scripts/backup-keyvault.sh
COPY src/avamar/azure/CLI/restore-keyvault.sh /dockerclient/etc/scripts/restore-keyvault.sh
RUN chmod 755 /dockerclient/etc/scripts/*.sh
# Housekeeping
RUN rm -rf /tmp/*
```

We can conclude the following after reading the dockerfile generated by the pre-build function:

- The **cloud provider** is Azure because this file installs az-cli.
- The **use case** is key vault.
- The **backup platform** is Avamar plus Data Domain because this file installs the Avamar-Client rpm file and the DDBoostFS rpm file.
- We only know the **access method** once we review the backup or restore script that we have included in the container; in this case, we will see the restore script called **restore-keyvault.sh**.

```
#!/bin/bash
version="1.0.0"
set -euo pipefail
if [ ! -z ${1} ]; then AKVRESTORE=${1}; else echo Missing keyvault, exiting; fi

vaultToken=$(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 | sed -n 's|.*"access_token": *"\([^"]*\)".*|\1|p')

secrets=$(shopt -s nullglob dotglob; echo $RestoreDir/secrets/*)
if (( ${#secrets} ));
then
    for i in $(ls $RestoreDir/secrets/*); do
        echo "*** Restore of secret $i in KeyVault $AKVRESTORE ***"
        response=$(curl --location --request POST "https://${AKVRESTORE}.vault.azure.net \
            /secrets/restore?api-version=7.3" \
            --write-out %{http_code} \
            --header "Authorization: Bearer ${vaultToken}" \
            --header "Content-Type: application/json" \
            --data "@$i")
        if [ ${response:2:5} == "error" ]
        then
            echo Restore of secret from file $i is not successful
            exit 1
        fi
    done
fi
```

Then:

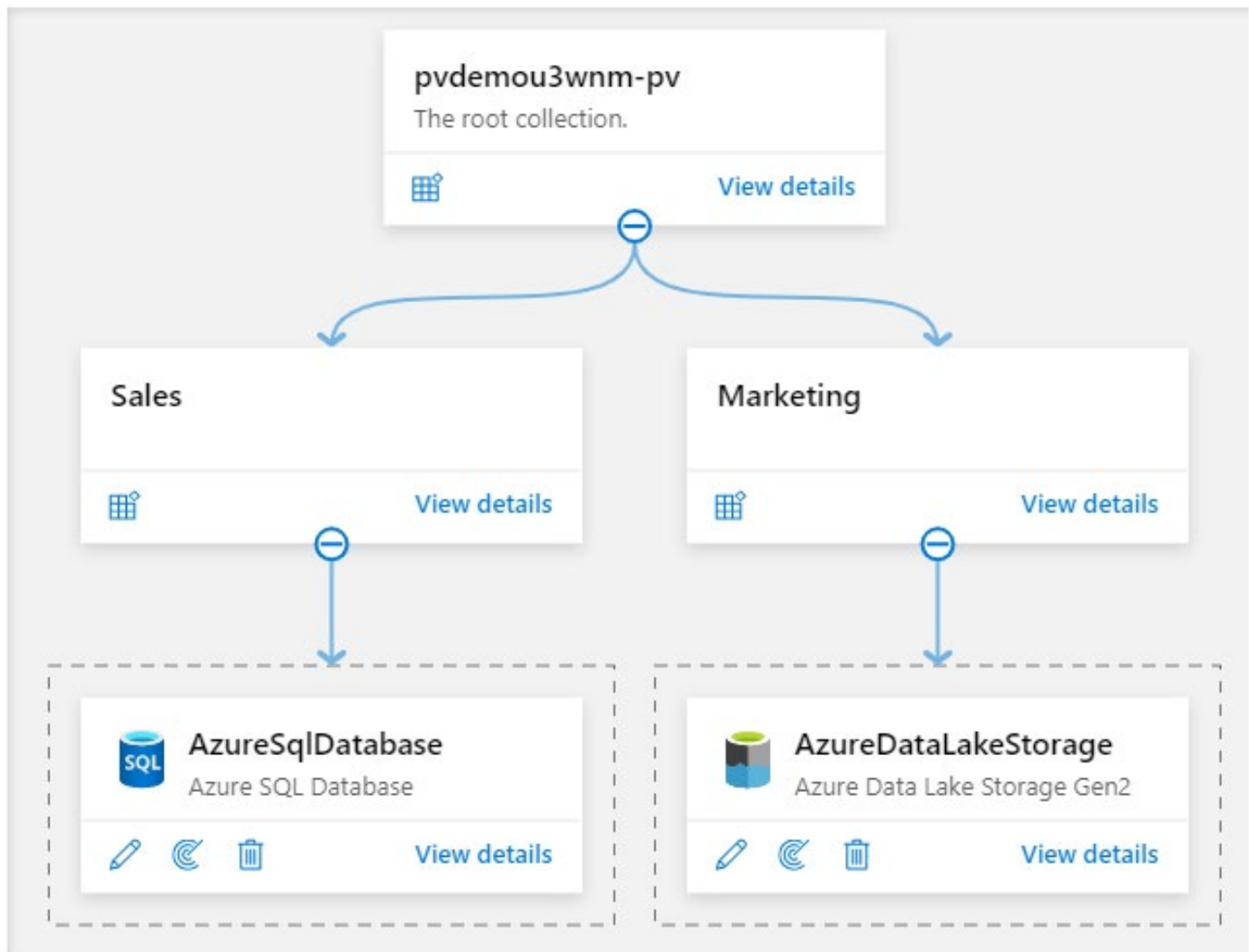
- The curl command denotes that the **access method** is API.

Example 3: *“We want to back up a complex use case that includes database-based, file-based, and special use cases.”*











We will use Microsoft Purview Demo Environment# to get a pre-populated Microsoft Purview account.

Microsoft Purview provides a unified data governance solution to help manage and govern your on-premises, multi-cloud, and software-as-a-service (SaaS) data. Easily create a holistic, up-to-date map of your data landscape with automated data discovery, sensitive data classification, and end-to-end data lineage. Enable data consumers to access valuable, trustworthy data management#.

After deploying the template, three collections and two sources were deployed:



The underlying technologies used to create this data governance service are composed by:

<input type="checkbox"/> Name ↑↓	Type ↑↓
<input type="checkbox"/>  configDeployer	Managed Identity
<input type="checkbox"/>  pvdemoukeld-adf	Data factory (V2)
<input type="checkbox"/>  pvdemoukeld-keyvault	Key vault
<input type="checkbox"/>  pvdemoukeld-pv	Microsoft Purview account
<input type="checkbox"/>  pvdemoukeld-sqldb (pvdemoukeld-sqlsvr/pvdemoukeld-sqldb)	SQL database
<input type="checkbox"/>  pvdemoukeld-sqlsvr	SQL server
<input type="checkbox"/>  pvdemoukeld-synapse	Synapse workspace
<input type="checkbox"/>  pvdemoukeldadls	Storage account
<input type="checkbox"/>  pvdemoukeldsynapsedl	Storage account
<input type="checkbox"/>  script	Deployment Script

In a simplified way:

- Microsoft Purview Account
- Azure Key Vault
- Azure Data Lake Storage Gen2 Account
- Azure Data Factory
- Azure Synapse Analytics Workspace
- Azure SQL Database

We must protect the data generated and stored in all these technologies, plus the full definition of Purview.

To simplify this use case, which is quite complex, we have deployed the Purview account in an Azure resource group explicitly created for this purpose; this is not mandatory. The resource group is called "purview."

We will execute an AZ CLI command that returns all the types of resources and their names to know what to protect.

```
az resource list --resource-group purview | jq -r '(.[] | [.type, .name]) | @tsv'
```

The output of the previous command will be like the following:

```
Microsoft.Storage/storageAccounts      pvdemocvhfwsynapsed1
Microsoft.ManagedIdentity/userAssignedIdentities  configDeployer
Microsoft.Purview/accounts             pvdemocvhfw-pv
Microsoft.Storage/storageAccounts      pvdemocvhfwad1s
Microsoft.Sql/servers                   pvdemocvhfw-sqlsvr
Microsoft.Sql/servers/databases        pvdemocvhfw-sqlsvr/pvdemocvhfw-sqldb
Microsoft.Sql/servers/databases        pvdemocvhfw-sqlsvr/master
Microsoft.Synapse/workspaces           pvdemocvhfw-synapse
Microsoft.KeyVault/vaults              pvdemocvhfw-keyvault
Microsoft.DataFactory/factories        pvdemocvhfw-adf
Microsoft.Resources/deploymentScripts  script
```

Now that we have all the Azure resources that Purview needs, we must safeguard them. The computer code to perform this task is extensive, and we will only transcribe the essential parts in this document.

The following code backs up the Kay Vault SQL secret and the reader can pre-learn the secret's name using one of these techniques: Labeling/resource scanning/Exhaustive enumeration.

```
if [ Microsoft.KeyVault/vaults ]; then

resource=KeyVault
curl --location --request POST \
    "https://$pvdemocvhfw-keyvault.vault.azure.net/secrets/${linea}/backup?api-version=7.3" \
    --header "Authorization: Bearer ${vaultToken}" -H "Content-Length: 0" \
    > ${BackupDir}/${resource}/secrets/$DOCKERNAME.pvdemocvhfw-keyvault.S.$(date +%Y%m%d%H%M%S).bkp

fi
```

In the case of the storage account involved, we can mount its containers with blobfuse or rclone; in our example, it will be with blobfuse.

```
if [ Microsoft.Storage/storageAccounts ]; then

resource=storageAccounts
pass="$(az storage account keys list --account-name pvdemocvhfwadls --query "[].{value:value}" \
--output tsv | head -1)"
containers="$(az storage container list --account-name ${1} --account-key {$pass} \
--query "[].{name:name}" --output tsv)"

for container in ${containers[@]}; do
blobfuse ${BackupDir}/${resource}/${1}/${container} \
--tmp-path=/tmp/blobfusetmp.pvdemocvhfwadls.${container} \
-o attr_timeout=240 \
-o negative_timeout=120 --config-file=${ConfigDir}/${container}.authspn \
--log-level=LOG_WARNING \
--pre-mount-validate=true --file-cache-timeout-in-seconds=120 -o ro -o nonempty
do
fi
```

We are going to protect the Data Factory pipelines. For them, it is required that DF be configured to generate a copy of them on a GitHub repository.

```
if [ Microsoft.DataFactory/factories ]; then

resource=DataFactory
repoConfiguration.accountName=pvdemocvhfw-adf

git clone --single-branch -b <repoConfiguration.collaborationBranch> \
https://<personalAccessToken>@dev.azure.com/<repoConfiguration.accountName>/ \
<repoConfiguration.repositoryName>/ \
_git/<repoConfiguration.repositoryName>

fi
```

There are several techniques to protect an AZ SQL database, including using SQL Package or leaving a copy in a blob account; we will use the second technique implemented in a Rest API call.


```

if [ Microsoft.Sql/servers ]; then

resource=Sql
db=$(curl --location --request GET \
  "https://management.azure.com/subscriptions/$SUSCRIPTIONID/resourceGroups/$RESOURCEGROUP/ \
  providers/Microsoft.Sql/servers/$linea/databases?api-version=2022-05-01-preview" \
  --header "Authorization: Bearer $mgmtToken" | jq -r '.value | .[].name' | grep -v master)

curl --location --request POST \
  "https://management.azure.com/subscriptions/$SUSCRIPTIONID/resourceGroups/$RESOURCEGROUP/providers/ \
  Microsoft.Sql/servers/pvdemocvhw-sqlsvr/databases/$db/export?api-version=2022-05-01-preview" \
  --header "Authorization: Bearer $mgmtToken" \
  --header "Content-Type: application/json" \
  --data-raw "{
    \"storageKeyType\": \"StorageAccessKey\",
    \"storageKey\": \"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\",
    \"storageUri\": \"https://<sta>.blob.core.windows.net/<container>/pvdemocvhw-sqlldb.bacpac\",
    \"administratorLogin\": \"<user>\",
    \"administratorLoginPassword\": \"<password>\",
    \"authenticationType\": \"Sql\" }"

fi

```

We remember the name and type of the Managed Identity.

```

if [ Microsoft.ManagedIdentity/userAssignedIdentities ]; then

resource=ManagedIdentity
az identity show --resource-group purview --name configDeployer > \
  ${BackupDir}/${resource}/configDeployer-purviewManagedIdentity.configDeployer.json

fi

```

Finally, the Purview backup script will be based on <https://github.com/tayganr/purviewcli>, which provides a command line interface to Microsoft Purview's REST API.

In the taken portion that we are going to show, we list all the Purview accounts of the resource group (in ours and having chosen a simplified example, it is only one), and for each account, all the collections that exist (in the model there are two and they are called "Sales" and "Marketing")

```

if [ Microsoft.Purview/accounts ]; then

resource=Purview
accountNames=$(pv management readAccounts --subscriptionId=$SUSSCRIPTIONID | jq -r '.value[].name')
collections=$(pv account getCollections | jq -r '.value[].name')

for accountName in $accountNames
do
    if [ ! -z "$collections" ]; then
        for collection in $collections
        do
            pv account getCollection --collectionName=$collection > \
                ${BackupDir}/${resource}/${accountName-collection-$collection.json}
        done
    fi
done
fi

```

Purview definitions

The complete code includes the following components:

- Keys for SHIR
- Private endpoint definitions
- Operations and keys definitions
- Glossary and terms definitions
- Resource Set Rules
- Policies definitions
- Classification Rule definitions
- Data sources definitions
- Scans and Scan Rulesets definitions

The reader will notice that we have created a platform that allows, after a use case design (**), the combination of different cloud providers with use cases, backup platform, and access methods; components that will enable us to form a four-dimensional matrix and multiple combination possibilities, some of them will be trivial and should be discarded, but we will have many options.

Following there is a list of examples, with a couple of alternatives that are feasible to deploy

Container software



Use case	Cloud provider	Backup platform	Access method	Package included in container
Purview	▼ Azure	Avamar/DD	API	Azure client, Avamar client, DDboost client
S3	▼ AWS	Avamar	CLI	AWS client, Avamar client, S3 fuse client
MinIO	▼ Azure	Networker	CLI	Azure client, Networker client, RClone client, MinIO Client
Cloud Storage	▼ GCP	Networker	CLI	GCP client, Networker client, S3 fuse client
Blob Storage	▼ Azure	Networker	CLI	Azure client, Networker client, Azure fuse client
Azure SQL	▼ Azure	Avamar/DD	CLI	Azure client, Avamar client, DDboost client, SQLPackage
PostgreSQL	▼ AWS	Avamar/DD	CLI	AWS client, Avamar client, DDboost client, PostgreSQL client
MariaDB	▼ Azure	Avamar/DD	CLI	Azure client, Avamar client, DDboost client, MySQL client
Keyvault	▼ Azure	Avamar/DD	API	Azure client, Avamar client, DDboost client
Kafka	▼ Azure	Avamar/DD	CLI	Azure client, Avamar client, DDboost client, Legacy backup script
File Storage	▼ Azure	Networker	CLI	Azure client, Networker client, CIFS client
Cosmo MongoDB	▼ Azure	Avamar/DD	CLI	Azure client, Avamar client, DDboost client, MongoDB client
DynamoDB	▼ AWS	Avamar/DD	CLI	AWS client, Avamar client, DDboost client

(**) There is no methodology to create a use case; they are very dissimilar because some are very simple, such as blob-based storage; others are moderately complex, such as a database; and finally, some involve the interrelation of many components, such as Azure Purview; anyway, the following is a guide you can follow to build a use case from scratch.

Step

1 Find the cloud provider that your client has chosen to consume the service	<i>AWS, Azure, and GCP</i>
2 Look for native cloud provider tools that extract all the resource properties	<i>mongodump, API Rest interface, FUSE driver</i>
3 Look for native backup tools that the service manufacturer has created.	<i>Kafka backup script is already created, Purview CLI.</i>
4 Determine if you need an intermediate repository to be accessed by the DDbost protocol.	
5 Create two automation scripts parameterized via JSON, YAML, or XML file	
6 Create a script to fill the service with test data	
7 Create a business JSON file with all the information required so that the prebuild function can create a docker	<i>Cloud client</i> <i>Backup client (and DDbost)</i> <i>Application backup or restore client if you do not use APIs</i> <i>Backup and Restore Scripts.</i>
8 Use the "build" function to create a docker image.	
9 Use some of the "deploy," "deploy-openshift," or "deploy-kubernetes" functions to push the container to the destination.	
10 Create a policy in your chosen backup software, setting the container as a client (for Avamar, this process is done automatically)	
11 Test	

Conclusion

In this article, we have developed a methodology with many examples of creating a "cloud-aware" layer that can be adapted to any cloud provider and potentially protect any service.

To do this, we have explored technologies commonly used to orchestrate cloud services, including GitHub, Docker, Podman containers, OpenShift and Kubernetes clusters, Shell scripts, and Docker and JSON files. We use examples of REST API calls and command line interfaces to log in and search for resources (services) in the cloud.

The cooperative and decentralized development of new services worldwide has imposed a new paradigm regarding the speed of creating resources that must be protected. Consequently, backup strategies and products must follow the same rules to be helpful.

The correct way to solve the challenge exposed in the previous paragraph is to develop a specific layer of software using collaborative tools that allow a reduction of the time-to-market of new services to almost zero while protecting without resigning all the characteristics that legacy backup applications developed over time. It is a new layer that connects to traditional backup products (or new ones if you wish). It is not necessary to solve functionalities that are already solved; the entire backup product does not need to be native, and a robust interface that negotiates with the cloud is sufficient. Due to operational costs, if the backup product is legacy, it is desirable to have a version installed in the cloud.

This architecture is resistant to ransomware simply by choosing a resistant backup platform.

Dell Technologies believes the information in this publication is accurate as of its publication date. The information is subject to change without notice. Disclaimer: The views, processes or methodologies published in this article are those of the authors. They do not necessarily reflect Dell Technologies' views, processes, or methodologies.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS." DELL TECHNOLOGIES MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying and distribution of any Dell Technologies software described in this publication requires an applicable software license.

Copyright © 2023 Dell Inc. or its subsidiaries. All Rights Reserved. Dell Technologies, Dell, EMC, Dell EMC and other trademarks are trademarks of Dell Inc. or its subsidiaries. Other trademarks may be trademarks of their respective owners.

Appendix

This appendix documents the most important properties to consider when creating a new use case; be creative; there are thousands of possibilities to configure and start saving data.

dockerType: docker type; this is the unique name of the use case.

cloudProvider: Cloud service provider.

useDDBoost: If the use case requires saving the information in Data Domain before being protected by the backup application.

useBlobFuse: For the different types of storage, indicates the kind that will use to mount the storage as a file system.

useCommand: Mutually exclusive with useAPI; if the data extraction is going to be done with a command.

useAPI: Mutually exclusive with useCommand; if the data extraction is going to be done by calling the REST API.

packages: Additional packages for each use case must be installed in the container.

backupContent (data/config): If the use case involves backing up the generated data, the service configuration, or both.

The reader can find some of the use cases configured in the following table.

dockerType	azsql	blobstorage	cosmosmg	datafactory	eventhub	filestorage	keyvault
cloudProvider	azure	azure	azure	azure	azure	azure	azure
useDDBoost	yes	no	yes	yes	no	no	yes
useBlobFuse	no	yes	no	no	yes	no	no
useCommand	yes	yes	yes	yes	yes	yes	yes
useAPI	no	no	no	no	no	no	no
packages	azsql	blobfuse	mongodb-datab	git	no	cifs-utils	no
backupContent	data	data	data	data	data	data	data
dockerType	kafka	minio	netapp	S3	elasticsea	redis	
cloudProvider	azure	azure	azure	aws	aws	aws	
useDDBoost	yes	no	no	no	no	no	
useBlobFuse	no	yes	no	yes	yes	yes	
useCommand	yes	yes	yes	yes	yes	yes	
useAPI	no	no	no	no	no	no	
packages	no	s3fs-fuse	nfs-utils	s3fs-fuse	no	no	
backupContent	data	data	data	data	data	data	

JSON Keys

- **cloudProvider** Azure/AWS/GCP
- **dockerType** adls/atlas/azsql/blobstorage/cosmosql/cvision/databriks/datafactory/eventhub/files storage/hdinsight/keyvault/kafka/cosmosmg/minio/netappstorage/postgresql/purview/redis
- **keyVaultName** Azure Key Vault name

- **useKeyVaultSecureAccess** Keyvault access using curl (YES) or az cli (NO)
- **useTags** tags or default values using fixValues
- **useFQDN** FQDN or IP through nslookup
- **cloudconnection** Proxies, certificates and end points
 - proxy
 - **useProxy** YES if docker file needs proxy ENV variables otherwise NO
 - **proxyHttpName and proxyHttpsName** Proxies FQDN and port values
 - **noProxy** No proxy for FQDNs (comma separated)
 - certs
 - **useCerts** YES if certificate is needed otherwise NO
 - **cers** Certificate name or * to include all
src/packages/DockerEmbebed/certificates/
 - EndPoints
 - **useEndPoints** YES if end points are used otherwise NO
 - **EndPoint** End point FQDN or IP
- **azureLogin** Resource group, identity and subscription
 - **resourceGroup** resource group name or all to all RGs
 - **tenantId** Tenantid
 - **ServicePrincipal**
 - **useServicePrincipal** YES for SPN otherwise NO
 - **servicePrincipalClientId** Service principal client id
 - **KeyVaultyes** If use key vault
 - **loginKeyVaultName** Key vault used to login
 - **secretSPN** Service principal client secret name
 - **KeyVaultno** Key vault is not used anymore
 - **servicePrincipalClientSecret** Hardcoded value
 - **ManagedIdentity**
 - **useUserAssignedRManagedIdentity** If user managed identity is used
 - **ManageldentityName** Management identity name
 - **useSystemAssignedManagedIdentity** If system managed identity is used
 - **Credentials**
 - **useCredentials** If credentials are used to login
 - **userName** User name. Two factor authentication is not supported
 - **Password** Password used to login
 - **subscription**
 - **changeDefaultsubscription** YES to change from default subscription
 - **subscriptionID** Subscription ID
- **containerName** FQDN of container used to register this client on Avamar. Add forward and reverse DNS records to DNS Server
- **azureResources** Azure specific information

- **resourceType** Azure resource type to be discover
- **useAutoDiscover** If autodiscovery feature is enabled
- **backupTags \ type** Type of tag, value examples: user/port/database/task/secret
- **backupTags \ value** Value of type tag
- **fixValues \ type** Type of tag, value examples: user/port/resource_list/task/secret
- **fixValues \ value** Hardcoded value