Terraforming Azure Data Services

Presented at Finland Microsoft BI & Power BI User Group

2.5.2020

What this talk will cover

- Brief introduction to Infrastructure-as-Code, what it is on Azure, and how Terraform fits into the picture
- Brief introduction to Terraform
- Demo - Creating a data processing infrastructure with Terraform on Azure, with demos
  - Architecture
  - Design considerations
  - Implementation in Terraform (and supporting tools when needed)

What this talk will NOT cover

- Terraform stuff above basic level (modularization, workspaces / environments, test autiomation)
- Terraform inner machinations
- Databricks to any great extent
- Spark

Agenda

- Brief introduction to Infrastructure-as-Code, what it is on Azure, and how Terraform fits into the picture
- Brief introduction to Terraform
- Demo - Creating a data processing infrastructure with Terraform on Azure
  - Architecture
  - Design considerations
  - Implementation in Terraform (and supporting tools when needed)

Infrastructure-as-Code (IaC)

What it is

- Deployable textual definition of your infrastructure (infrastructure here in the wide sense – IaaS, PaaS, SaaS)
- Especially usable in cloud environments, but many tools support on-premise environments as well
- Both imperative and declarative tools available
- Related concepts: immutable infrastructure, configuration-as-code, governance-as-code, X-as code

Benefits and considerations

Benefits:

- Reproducibility – easier to launch new environments or new projects with a defined baseline
- Easier to manage changes and version

Considerations:

- Always slower than just clicking away in the portal – take upfront work into account
- Reverse engineering a deployed environment to IaC is a major pain – start at the beginning

Tools on Azure

Azure-specific:

- Powershell
- Azure CLI
- Azure Resource Manager (ARM) templates
- Service-specific SDK's (e.g. Machine Learning Service)

Third-party:

- Ansible
- Chef
- Puppet
- Pulumi
- Terraform

Agenda

- Brief introduction to Infrastructure-as-Code, what it is on Azure, and how Terraform fits into the picture
- Brief introduction to Terraform
- Demo - Creating a data processing infrastructure with Terraform on Azure
  - Architecture
  - Design considerations
  - Implementation in Terraform (and supporting tools when needed)

Terraform - Overview

- Developed by HashiCorp, built on Go language, open source (codebase on GitHub)
- Currently at version 0.12.20, under rapid development
- Popular especially with AWS, but can well be used also in other environments (more on that in a bit)
- Own script language HCL (Hashicorp Configuration Language)

```
resource "azurerm_resource_group" "rg" {
    name     = "myTFResourceGroup"
    location = "eastus"

    tags = {
        environment = "TF sandbox"
    }
}
```

Terraform – Installation & Getting Started

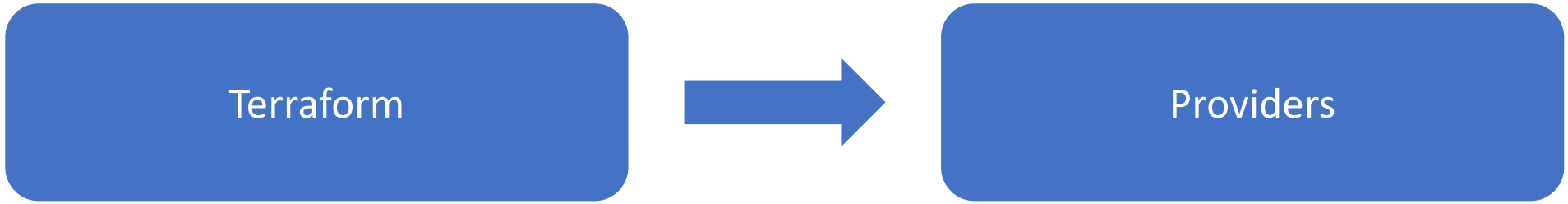
- **Installation:**
- Distributed as a single executable for Windows / MacOS / Linux
- Also available on MacOS and Linux package managers like Homebrew
- → Very easy to set up and update
- Link: https://www.terraform.io/downloads.html

- **Getting started:**
- Terraform has great materials to get started, for Azure, AWS and GCP: https://learn.hashicorp.com/terraform
- Some materials for more advanced use:
  - Modularization etc. (examples for AWS but concepts fit any provider): https://blog.gruntwork.io/a-comprehensive-guide-to-terraform-b3d32832baca
  - Environment management: https://www.hashicorp.com/resources/evolving-infrastructure-terraform-opencredo

Terraform – Main features

- Separation of Terraform tool and providers
- Storing of information on the deployment state
- Workflow: Validate – plan – apply

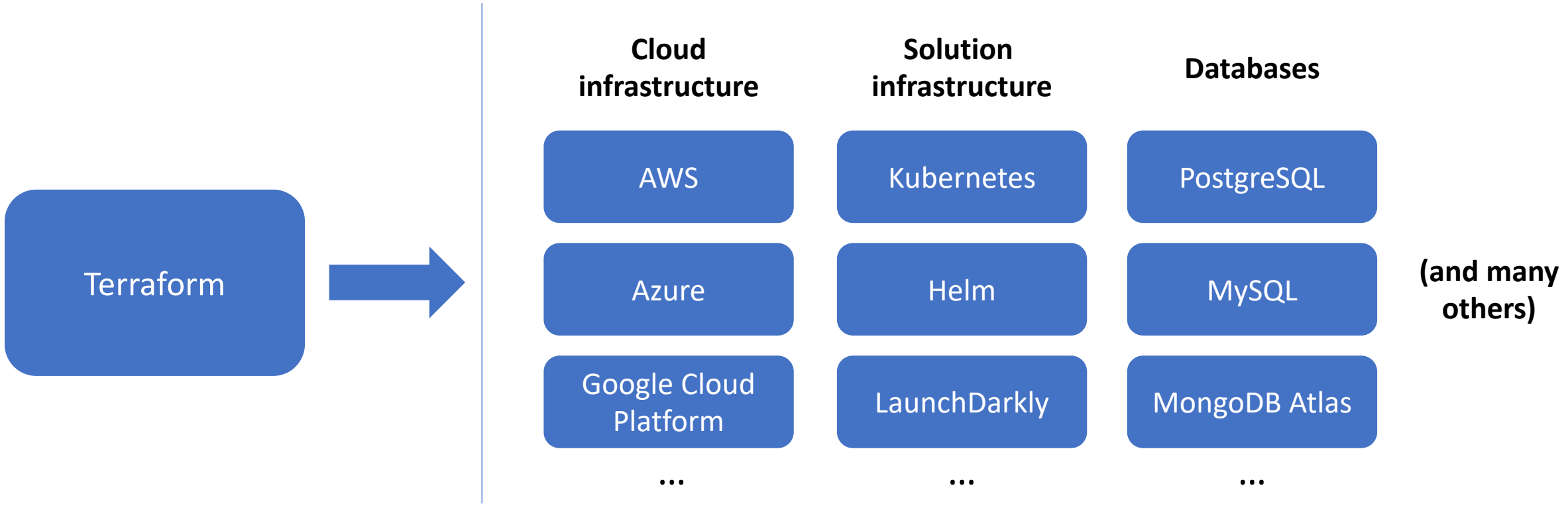
Main feature 1 - Separation of Terraform tool and providers



- Terraform is developed and supported mainly by HashiCorp
- Terraform provides the framework for infrastructure management
- The framework provides functionalities such as the configuration language itself managing the state, forming deployment plans, etc.

- Terraform providers are primarily developed by other parties who want to enable deployment of their services with Terraform (though often in collaboration with HashiCorp)
- For example, there are providers for all the major clouds, for on-premise virtual machines, for Kubernetes, etc.

Main feature 1 - Separation of Terraform tool and providers



Due to the distributed nature of provider development, development is rapid but quality can be uneven

Main feature 2 - State

- Terraform stores all information on the deployments it manages in a **state file**
- The state file enables Terraform to check current code against the deployed environment, and target only the resources that have changes in code – **this is one of Terraform's main strengths**, as it makes deployments faster and effects more targeted
- **However, storing the state also brings some (large) concerns:**
  - If you lose the state file, Terraform loses all information on the deployed environment – do backups (e.g. soft delete when using Azure Storage as backend)
  - The state file contains every bit of information contained in your environment in plain text, which means every secret and key for resources you create with Terraform - keep the state file secure, and possibly have an external rotation logic running
  - When developing as a team, all team members need to have access to the same state file – luckily this is easy to achieve nowadays using a remote backend (more on that in a bit)

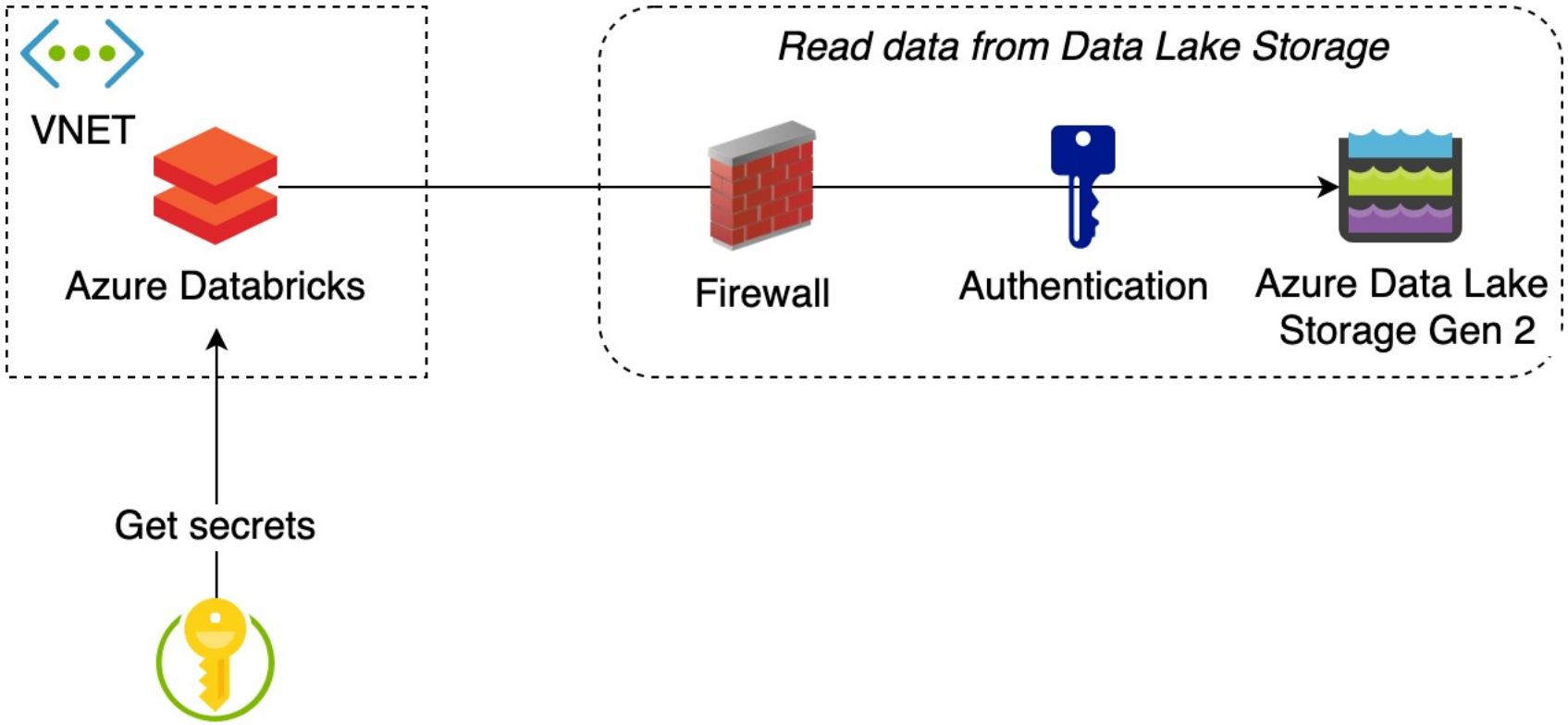
Main feature 3 – Validate – Plan – Apply

- Running Terraform should always be done in 3 stages:
- **Validate (*terraform validate*):**
  - Terraform runs static validation for your code – catches e.g. misnamed variables and such
  - However, cannot catch errors that are only surfaced upon deployment – e.g. naming collisions or non-allowed cloud-specific settings
  - Validation is always run as part of *terraform plan*, but can be run separately as well
- **Plan (*terraform plan*):**
  - Terraform compares the known state and current code, and creates a plan for what to deploy (created as a directed acyclic graph)
  - Plan is provided for analysis before apply
  - Plan phase is run as part of apply by default
- **Apply (*terraform apply*)**
  - Applies the changes in the previously created plan

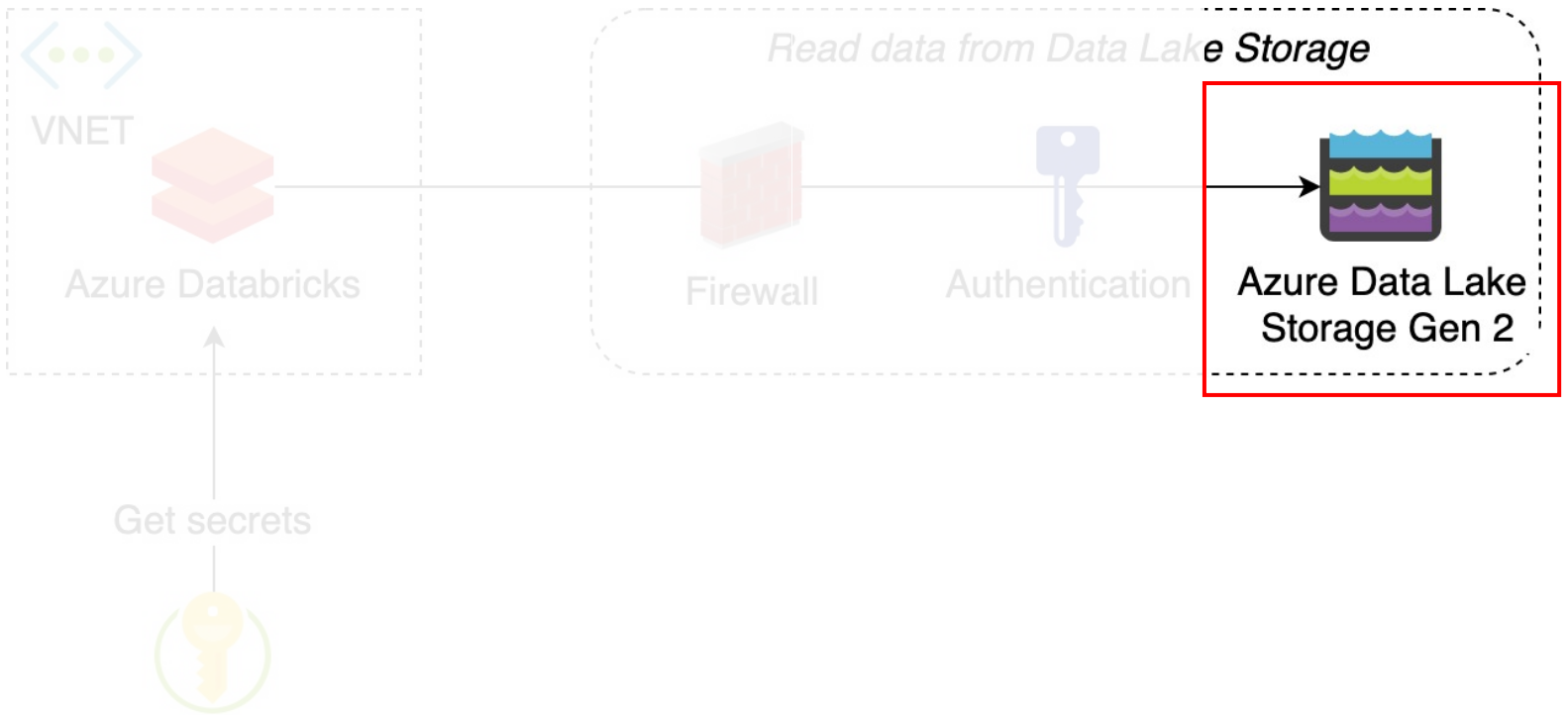
Agenda

- Brief introduction to Infrastructure-as-Code, what it is on Azure, and how Terraform fits into the picture
- Brief introduction to Terraform
- Demo - Creating a data processing infrastructure with Terraform on Azure
  - Architecture
  - Design considerations
  - Implementation in Terraform (and supporting tools when needed)

Overall architecture



Step 1 – Data Lake Storage



Azure Data Lake Storage Gen 2

- Extension of Storage Account
- Adds hierarchical namespace which speeds data reads
- Has own firewall settings (also can be deployed really to a VNET with Private Link)
- With Databricks, supports authentication with Storage Keys and with Service Principal

Step 2 – Databricks with VNET injection

Azure Databricks

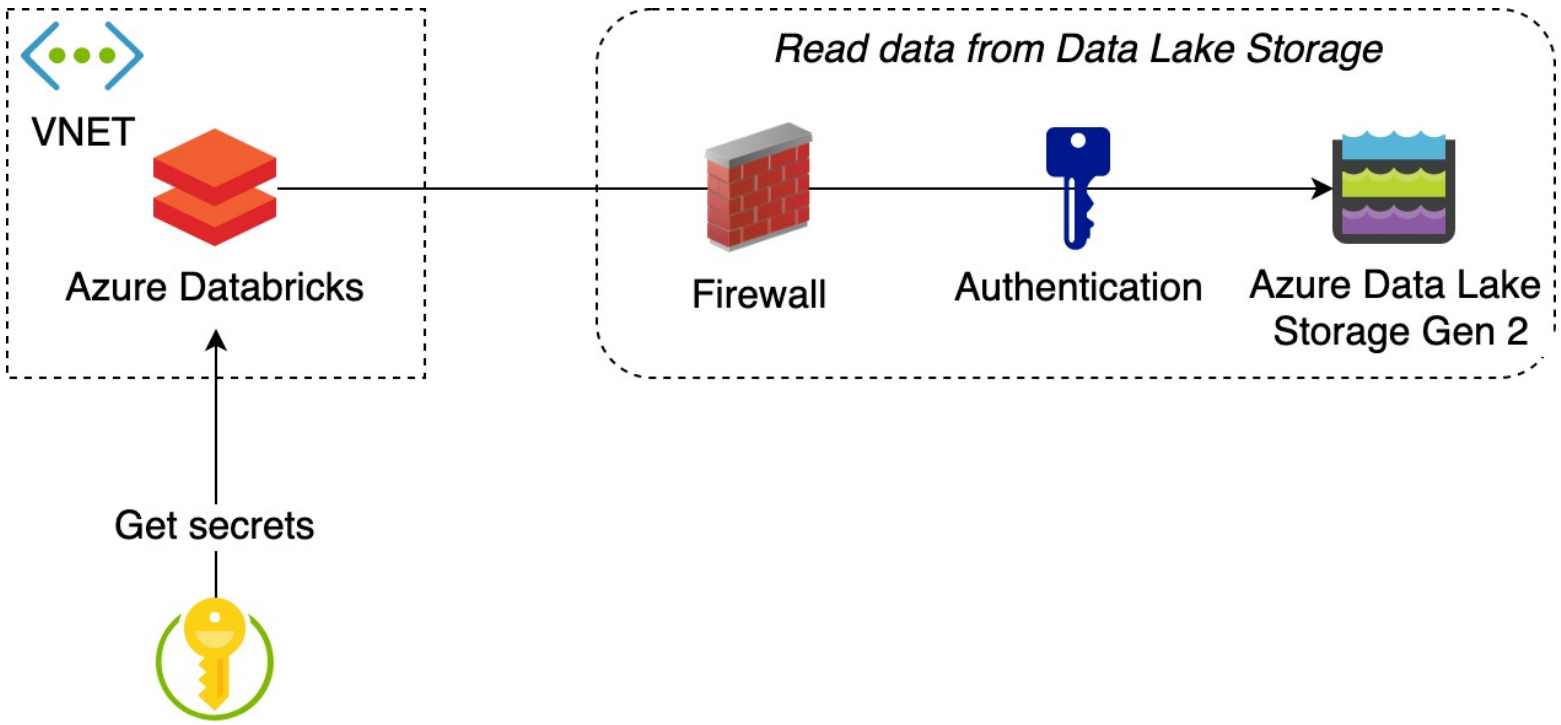
- Scalable data processing tool based on Apache Spark
- In order to connect to firewall-protected services, you to place the Databricks clusters to a VNET
- Requires 2 dedicated subnets, public and private
  - Public is for connections from cluster nodes to external services
  - Private is for cluster internal communications
  - Each cluster node is connected to both subnets

Step 3 – Key Vault

Azure Key Vault

- Service to store secrets (keys, certificates)
- Can be added as a secret scope to Databricks
- After this, Key Vault secrets can be fetched into Databricks processes
- This is a great way to store e.g. access credentials and environment variables

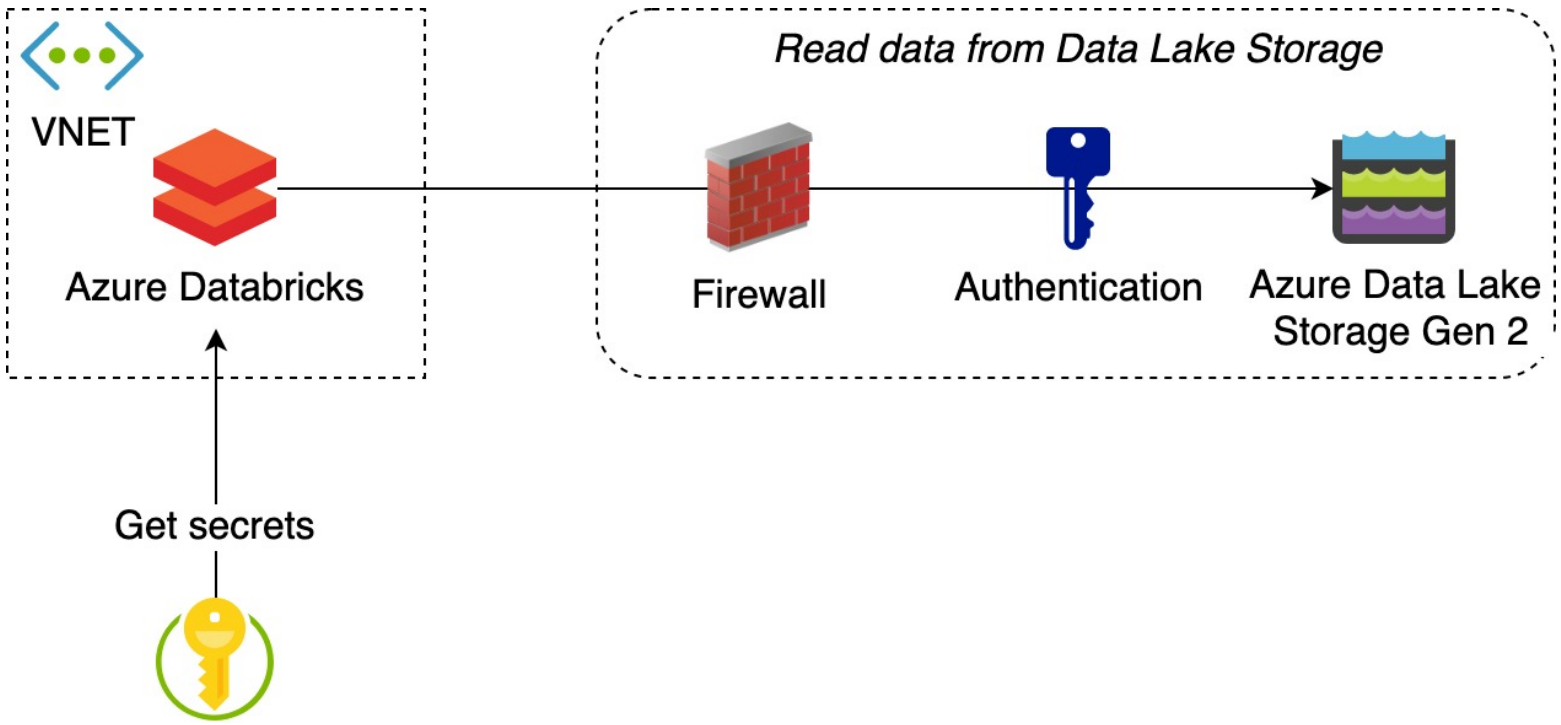
Step 4 – Authenticate to ADLS with Storage key



Authentication method 1

- Authentication to Data Lake Storage from Azure Databricks using Storage key

Step 5 (if enough time) – Authenticate to ADLS with Service Principal



Authentication method 2

- Authentication to Data Lake Storage from Azure Databricks using Azure AD Service Principal