

XPR.T

Magazine N°17/2024

Revealing Strength: Ignite Capabilities

Managed DevOps Pools

Kubernetes Network Policy

Stop Creating Content With ChatGPT

Democratizing access to AI through
GitHub Models

Generating Documentation With
Azure AI

Xebia



DO NOT
DISTURB

We Are Xebia

Your Trusted Microsoft
Solutions Partner



GitHub Verified Partner

Xebia

xebia.com

epic

Microsoft
Solutions Partner
Infrastructure
Azure

Microsoft
Solutions Partner
Security

Microsoft
Solutions Partner
Digital & App Innovation
Azure

Microsoft
Solutions Partner
Data & AI
Azure

Specialist
Kubernetes on Azure
Accelerate Developer Productivity
with Microsoft

Specialist
Kubernetes on Azure

Colophon

XPRT. Magazine N°17/2024

Editorial Office

Xebia

This magazine was made

by Xebia

Michiel van Oudheusden, Rob Bos,
Patrick de Kruijf, Jelmer de Jong,
Matthijs van der Veer, Loek Duys,
Chris van Sluijsveld, Sander Aernouts,
Arjan van Bekkum, Robert de Veen,
Rutger Buiteman, Marcel de Vries

Contact

Xebia
Laapersveld 27
1213 VB Hilversum
The Netherlands
Call +31 35 538 19 21
xmsinfo@xebia.com
www.xebia.com

Photography

Kim Ellermann

Layout and Design

Studio OOM / www.studio-oom.nl

Translations

Mickey Gousset (GitHub)

© Xebia, All Right Reserved

Xebia recognizes knowledge exchange as prerequisite for innovation. When in need of support for sharing, please contact Xebia. All Trademarks are property of their respective owners.



GitHub Verified Partner

If you prefer the digital version of this magazine, please scan the qr-code.



This issue of **XPRT.** magazine is about Revealing Strength: Ignite Capabilities.

Intro

004 XPRT. Magazine

Power Through Platforms

005 Ensuring Effective Cost Allocation in Azure

009 Managed DevOps Pools

017 Kubernetes Network Policy

023 How to build a maintainable and highly available Landing Zone

034 Stop Creating Content With ChatGPT

039 Democratizing access to AI through GitHub Models

State-of-the-Art Software Development

043 Was Shift Left The Right Move

047 The Future of Cloud-Native Software Development with Radius

Smooth Delivery

055 Generating Documentation With Azure AI

XPRT. Magazine

It's remarkable to reflect on how much has changed since our first magazine edition, which was heavily focused on software development methodologies and practices. Back then, we were all about refining how we build software, making our processes more efficient, and improving team collaboration. Fast forward to today, and the landscape has shifted quite a bit. Together with software development, we also dive deep into AI, cloud infrastructure, and automation. The focus has expanded beyond development to the tools and platforms that make everything run smoother, smarter, and faster.

Author Marcel de Vries

What fascinates me most is how AI and infrastructure have become foundational to almost everything we do. It's no longer just about writing better code or writing cloud-native code; it's about building systems that have the potential to optimize themselves, scale on demand, and handle complex workflows with minimal human intervention. AI transforms how we think about automation, from generating documentation to powering large-scale development environments like GitHub Copilot. Meanwhile, the infrastructure side has become a critical piece of the

puzzle—cloud-native tools, network policies, and cost management are no longer just "nice-to-haves" but essential components of any modern tech stack.

This shift in focus—from code-centric to platform- and AI-centric—speaks to how much opportunity we have at our fingertips now. By embracing these technologies, we're not just building software faster, cheaper, and better; we're creating ecosystems that empower businesses to move faster, innovate more, and tackle challenges that were

unthinkable just a few years ago. It's an exciting time to be in tech, and I'm thrilled to see where these evolving capabilities will take us next. I am also proud of the team that put this magazine together again. It is such a blessing to be working with a team that constantly pushes the boundaries of what is possible and embraces the constant change coming to us. With this magazine, we share what we have learned and our insights, and I hope we can inspire you to do the same in your organization!



Ensuring Effective Cost Allocation in Azure

I have been in organizations where we received a monthly email with a massive Excel spreadsheet, asking who owns which cost and how we could reduce it. As you can imagine, this process repeated itself because no one knew or claimed ownership. This is why it is crucial that the costs reported by the cloud provider and visible on the cloud bill can be linked to the right owners.

Author Michiel van Oudheusden

Cost allocation is the set of practices used to divide up a consolidated cloud invoice among those responsible for its various components. In the context of FinOps, this involves dividing the cloud service provider invoices among the different IT groups within an organization.

Why is this important? Cost allocation is essential for showback and chargeback. By bringing the costs back to individual projects or teams within an organization, we achieve financial transparency and accountability. This, in turn, helps optimize costs, as it becomes clear who owns what resources and who is responsible for managing and reducing expenses.

Cost allocation is achieved through a combination of functional activities, primarily focusing on using a consistent hierarchy of accounts, projects, subscriptions, resource groups, and other logical groupings of resources. This also includes applying resource-level metadata—tags or labels—within the cloud service provider or through a third-party FinOps platform.

Cost Allocation in Azure

Azure resources are always part of a resource group. Resource groups belong to a subscription, which in turn roll up into management groups. This tree-like structure provides a flexible framework for setting up various cost allocation schemes.

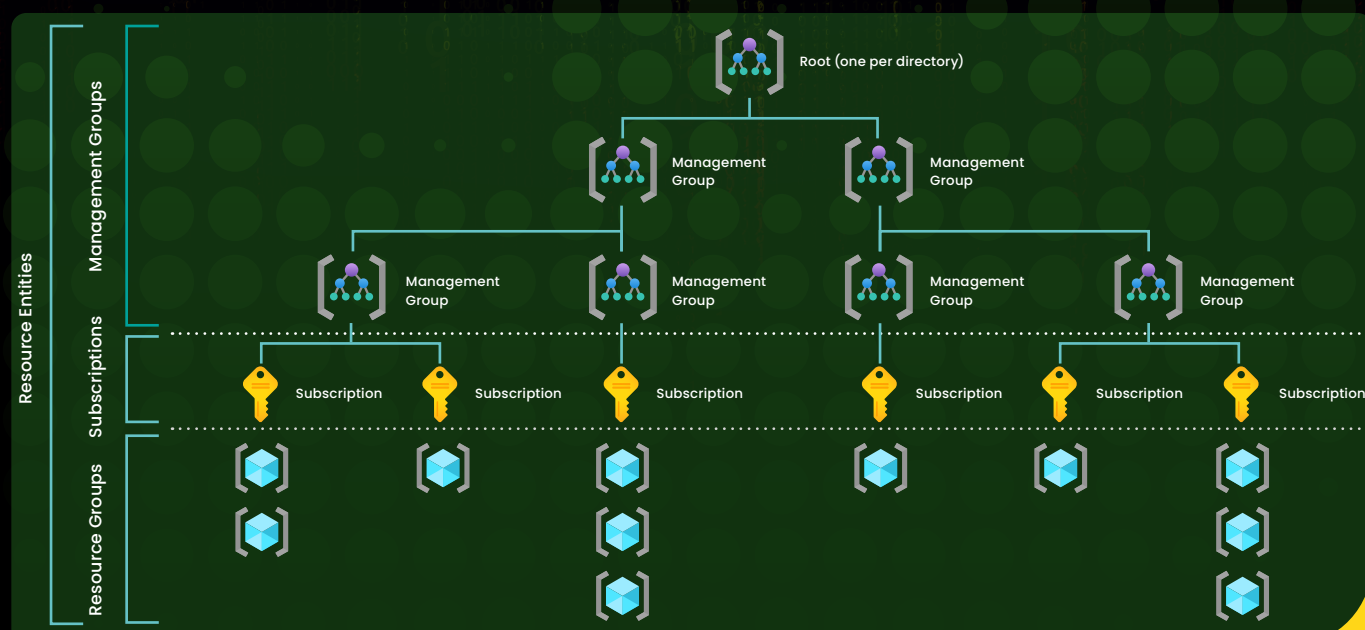


Figure 1: Azure Resource Entity Hierarchy and Management Groups^[1]

¹ <https://learn.microsoft.com/en-us/azure/cost-management-billing/costs/cost-allocation-introduction>

Management Groups

Management groups can be used to map subscriptions to different business units or environments. This top-level organization helps in categorizing costs at a broad level.

Subscriptions

Subscriptions are a crucial partition in Azure. They can be used for access control and governance, but also serve as a fundamental unit for cost allocation. Each subscription can be assigned to a specific department, project, or environment, allowing for clear visibility into spending.

Resource Groups and Tags

Within a subscription, resource groups can contain multiple resources. Resource groups support tagging, which involves assigning key/value pairs to resources. Tags can be used to specify details like project name, business unit, or cost center.

Tag Inheritance

By utilizing the inheritance feature^[2], tags applied to a resource group can be inherited by the resources within the group. This ensures consistent tagging across all resources, including the ones which do not expose tags, simplifying cost allocation and reporting.

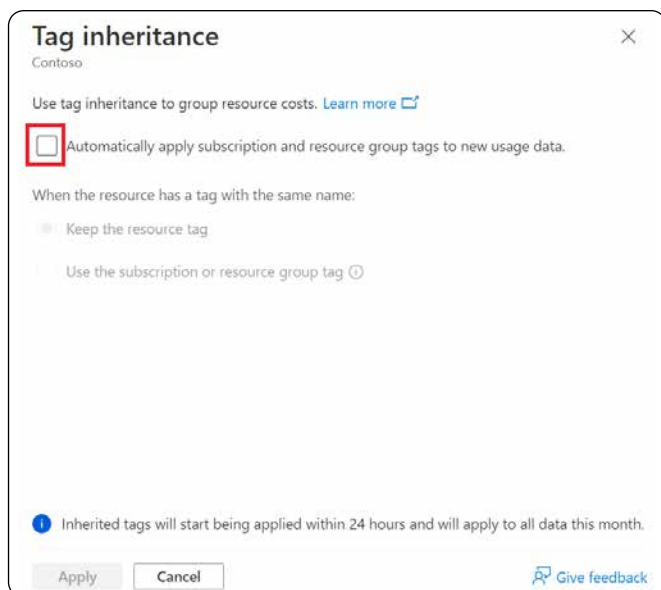


Figure 2: Automatically subscription and resource group tags to new usage data

Tag inheritance helps in maintaining uniformity and reducing the administrative overhead of manually tagging each resource.

Enforcing Tags with Azure Policies

Azure Policies can be used to enforce the use of tags. Policies can ensure that resources are tagged correctly upon creation and can even block the creation of resources that do not comply with the tagging strategy.

Azure Policies provide a mechanism to automate governance and ensure compliance with organizational standards, helping maintain a structured and manageable cloud environment.

By leveraging the hierarchical structure of management groups, subscriptions, and resource groups, along with effective tagging and enforcement policies, you can achieve precise cost allocation in Azure. This structure allows for detailed tracking and accountability, making it easier to manage and optimize cloud spending.

Shared Cost Allocation

In most organizations, systems are not operated in isolation but rely on shared components such as API Management, hub/spoke networks, virtual machines, and more. It would be unfair if the team delivering these shared services bore all the costs, while the teams consuming them paid nothing and perceived them as free services. Therefore, it's crucial to implement showback and potentially chargeback mechanisms for these shared costs.

Showback and Chargeback

Showback involves tracking and reporting the usage and associated costs of shared services to the consuming teams. **Chargeback** goes a step further, where the consuming teams are billed for their usage. Both practices promote financial transparency and accountability, ensuring that all teams are aware of and responsible for their consumption of shared services.

Configuring Cost Allocation Rules in Azure

If you are an Enterprise Agreement or Microsoft Customer Agreement user, Azure allows you to configure cost allocation rules. Here's how you can set it up:

Select Sources and Targets

- Identify the source subscriptions, resource groups, or tags representing the shared services whose costs need to be allocated.
- Identify the target subscriptions, resource groups, or tags representing the consuming teams or projects that should share these costs.

Determine Allocation Method

- **Even Distribution:** Spread the costs evenly across all targets.
- **Percentage-Based Distribution:** Allocate costs based on a predefined percentage for each target.
- **Usage-Based Distribution:** Allocate costs based on actual usage metrics such as compute, storage, or network consumption. This method ensures a fairer distribution of costs based on how much each team or project actually uses the shared services.

² <https://learn.microsoft.com/en-us/azure/cost-management-billing/costs/enable-tag-inheritance>



THE
NORTH
FACE

Targets	% TO ALLOCATE FROM SOURCE
ResourceGroup1	1
ResourceGroup2	38
ResourceGroup3	58
ResourceGroup4	3
Total	100%

Figure 3: Proportional to total cost

Important Considerations

While Azure allows you to configure cost allocation rules for showback purposes, it does not change the actual billing. If you want to implement chargeback, where consuming teams are billed for their usage, you will need to handle this outside of Azure using your organization's internal billing and accounting systems.

By implementing shared cost allocation, you ensure that all teams are aware of and accountable for their usage of shared services. This promotes a culture of cost-consciousness and helps optimize the overall cloud spending in your organization.

Validating and Analyzing Cost Allocation Setup

Now that you have created a cost allocation setup, whether through management groups, subscriptions, or tagging on resource groups, it's essential to validate and analyze the setup to ensure accuracy and effectiveness.

Validating Tagging with Azure CLI

To validate that your resources are correctly tagged, you can use the Azure CLI. The following command will list all the resource groups along with the "owner" or "creator" tags. If these tags are not set, it will indicate 'missing':

```
az group list --query "[{.name:name, Owner:tags.owner || tags.Owner || tags.creator || tags.Creator || 'missing'}]" -o table
```

This command will output a table showing each resource group with the specified tag names. Adjust the tag names as needed to match your organization's tagging conventions.

Using Azure Cost Analysis

Once your resources are tagged, the next step is to use Azure Cost Analysis to view and filter your costs:

Navigate to Azure Cost Management + Billing

- Go to the Azure Portal and select "Cost Management + Billing."
- Open the "Cost Analysis" section.
- Use the filtering options to filter costs by tags. This allows you to see the cost distribution based on the tags applied to your resources.

NOTE

Note that tags do not work retroactively^[3]. If a tag is applied to a resource today, it will not affect the cost data for previous dates. Ensure that tags are applied to the resources themselves, not just the resource groups. Use tag inheritance or Azure Policies to enforce consistent tagging across all resources.

Limitations and Considerations

While Azure's native tools provide powerful features for cost allocation, they do have some limitations:

- **Non-Retroactive Tags:** Tags applied today will not impact historical cost data.
- **Complex Filtering:** Azure's native tools may not support complex filtering and multi-dimensional cost allocation needs.

These limitations might warrant the use of third-party tools, which offer more advanced capabilities, such as:

- **Backdating Resources:** Ability to apply tags and allocate costs for historical data.
- **Multi-Dimensional Analysis:** Advanced filtering options to allocate costs based on multiple dimensions and criteria.

By validating your tagging setup and using Azure Cost Analysis, you can ensure accurate and effective cost allocation. This process helps maintain financial transparency and accountability, allowing for better cloud cost management and optimization.

³ <https://learn.microsoft.com/en-gb/azure/cost-management-billing/costs/understand-cost-mgt-data#how-tags-are-used-in-cost-and-usage-data>

Managed DevOps Pools

Have you ever had to deploy, configure and maintain your own DevOps agents, be it for Azure DevOps or GitHub, then you probably found out it is such a hassle to keep everything up-to-date and up-and-running.

Managed DevOps Pools have recently been announced as Public Preview. In this article we go over the most important features and capabilities of the new service, and will provide examples on how to implement this using Infrastructure as Code with Terraform.

Author Patrick de Kruijf

Managed DevOps Pools, what are they?

Managed DevOps Pools are Microsoft-hosted agents that can be configured to use private networking. This allows the agents to use private DNS zones, private endpoints, your own Azure Firewall (or an appliance) and with the added benefit of having Microsoft maintain these resources.

The Managed DevOps Pools also allow you to specify which SKU, Disks and OS Images you want to use. So whether you need specific compute power, a lot of disk space, or to use a specific OS Image (even from the Azure Marketplace), Managed DevOps Pools enable you to customize the agent pool to your needs.

There are two options to use when it comes to private networking. It can be configured to use an isolated network, which is supplied and managed by Microsoft, or it can use an existing virtual network within the Azure tenant.

The added benefit of the latter is that you are able to configure routing, DNS server configuration, and Network Security Groups, managing allowed and denied traffic more specifically for your needs.

There is another option that is quite powerful: you can use the Managed DevOps Pools for multiple Azure DevOps organizations and/or multiple projects within these organizations. Some companies use multiple DevOps organizations, i.e. one [1] for production and one [1] for sandbox environments. These can then all use the same Managed DevOps Pool.

So what's the managed part then?

Although you have to deploy some of the Azure resources yourself, the compute instances behind the scenes are managed by Microsoft. This will actually save you a ton of time and headaches (in my case). The downside of this behind the scenes management is you will not be able to see parts of the solution and therefore troubleshooting might become harder.

Summary

The benefits of Managed DevOps Pools are that while Microsoft will manage them, they are directly usable within your private networking. It has access to lots of images to be used for the agents, be it the Microsoft images, marketplace images, or even custom created images.

Three key takeaways for using Managed DevOps pools

By leveraging Managed DevOps Pools, you will have more time to spend on tasks that provide business value, and you will get the same amount of ease and security as you would with self-hosted agents.

1. Focus on business value – This service enables me to focus on delivering my business value, instead of maintaining and managing my self-hosted agents. I can automatically use the latest Microsoft hosted agent versions, without having to checkout the repository and build my custom images based on the Microsoft image repository.

2. Simplified administrative tasks – Not worrying about the compute instances of the Virtual Machine Scale Set, since those are maintained and managed by Microsoft as a Platform-as-a-Service (PaaS) offering. Also, the Azure DevOps agent pool configuration is done by the Managed DevOps Pool creation, so there is no need to configure Azure DevOps or wait for the Azure DevOps administrator to help out.

3. Managed Agents but with private networking –

The Managed DevOps Pools can directly use an available virtual network in the Azure tenant, allowing for better access to other services without the need to open up any services to the public internet, as you would have to using Microsoft-hosted agents.

Implementing DevOps Pools using Infrastructure as Code and Terraform

Ok, so let's start with identifying every service that we need and how it all works together.

Our objective is to set up a Managed DevOps Pool that is able to use private networking, linked to our networking hub, and use the same (Ubuntu) image as the Microsoft-hosted agents.

We prefer to use Infrastructure as Code (IaC) to minimize human failure and to create solutions that can be built, changed and managed in a consistent and repeatable way. My preferred IaC language is Terraform, so we will be using Terraform to deploy the resources.

This means we will have to perform the following tasks:

1. **Basic terraform setup** – We need to initialize Terraform and the basic repository to be able to deploy the Azure resources
2. **Request or update Quotas** – Managed DevOps Pool quotas are set to 0 by default, so we will need to request a quota increase in order to use the Managed DevOps Pools
3. **Create a resource group** – All Azure resources must be deployed into a resource group, so we will create a resource group for the Managed DevOps Pool resources
4. **Create a DevCenter and create a project** – DevCenter is a collection of projects with similar settings. It can be used to supply catalogs with Infrastructure as Code templates, which are available for all projects in the DevCenter, as well as creating development environments for development teams to use

5. **Create a Virtual network, peering to the central hub and create a subnet** – To enable private networking, we will need to create a virtual network with a subnet

6. **Create the Managed DevOps Pool** – The Managed DevOps Pool will create the compute instances that can run the Azure DevOps jobs, and we will supply the correct resource configuration based on the steps before

Basic terraform setup

When using Terraform, we will need to supply the provider information and configuration. We will be using both the **AzureRM** and the **AzAPI** providers. Details on using the **AzAPI** provider are described in the sections where it applies.

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "4.0.1"
    }
    azapi = {
      source  = "Azure/azapi"
      version = "1.15.0"
    }
  }
}

provider "azurerm" {
  subscription_id = "{insert your subscription ID here}"
  features {}

  resource_provider_registrations = "Extended"
  resource_providers_to_register = ["Microsoft.DevCenter", "Microsoft.DevOpsInfrastructure"]
}

provider "azapi" {}
```

Notice the new annotation for resource providers¹, which is introduced in the latest **AzureRM** version.

Also notice the providers, which are required to be set on the subscription so the subscription is activated to use the resources within the subscription:

- The **Microsoft.DevCenter** provider allows the creation and usage of the DevCenter resource and the projects in the DevCenter.
- The **Microsoft.DevOpsInfrastructure** provider enables the subscription to create and deploy the Managed DevOps Pools.

Managed DevOps Pools Quotas

One important thing to understand before deploying, is that we will need to request a quota increase. These quotas are specifically for Managed DevOps Pools, so your normal Virtual Machine SKU quotas are not valid for the Managed DevOps Pools SKUs.

¹ https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs#resource_provider_registrations

You can request the quotas as you would normally do with other quotas, via the Azure Portal on the subscription page:

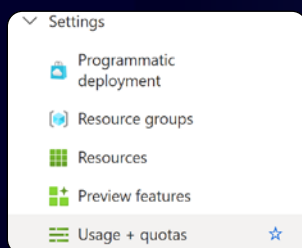


Figure 1: Usage + quotas

Make sure you select the Provider **Managed DevOps Pools** to see the quotas:

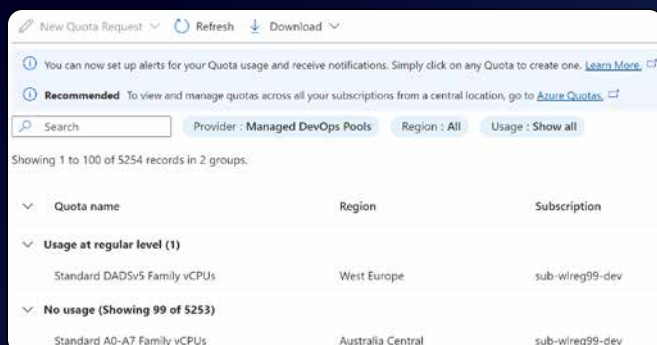


Figure 2: Managed DevOps Pools – quotas

If you do not know how to do this or if the button **New Quota Request** is greyed out, please reach out to your platform engineers or CSP to help you out.

The naming convention is provided in the **locals.tf** file.

```
locals {
  rgName      = "rg-${var.scaffold_company_short_name}-devpool-${var.scaffold_environment}-${var.scaffold_location_short_name}-001"
  vnetName    = "vnet-${var.scaffold_company_short_name}-devpool-${var.scaffold_environment}-${var.scaffold_location_short_name}-001"
  devCenterName = "devc-${var.scaffold_company_short_name}-${var.scaffold_environment}-${var.scaffold_location_short_name}-001"
  devCenterProjectName = "devpr-${var.scaffold_company_short_name}-devpool-${var.scaffold_environment}-${var.scaffold_location_short_name}-001"
  snetName     = "snet-${var.scaffold_company_short_name}-devpool-${var.scaffold_environment}-${var.scaffold_location_short_name}-001"
  poolName     = "pool-${var.scaffold_company_short_name}-devpool-${var.scaffold_environment}-${var.scaffold_location_short_name}-001"
}
```

NOTE

That the **devCenterName** value does not contain the **-devpool** section, this is done to stay within the naming length restriction of 26 characters. We have validations on the variables to ensure this naming convention cannot surpass the length restriction.

The locals provide the naming convention that I like to use for this solution. Feel free to change them to your needs or preference accordingly. The naming convention is based on the Cloud Adoption Framework naming convention². Feel free to overwrite these when other naming conventions should apply.

Variables and locals

I am using a **variables.tf** and **locals.tf** file to determine certain values to be used in the deployment, which I will briefly explain in this section.

The variables.tf describes all the variables that need to be supplied to run the Terraform deployment. The full file is available in the GitHub repository (available in the links section), but below two variables are shown:

```
variable "scaffold_company_short_name" {
  description = "Abbreviation of the company name to make all Azure resources unique within the Azure Tenant."
  type        = string

  validation {
    condition     = length(var.scaffold_company_short_name) <= 6
    error_message = "The company short name must be 6 characters or less."
  }
}

variable "devops_organization_url" {
  description = "The URL of the Azure DevOps organization to add the Managed DevOps Pool to."
  type        = string
}
```

These variables are used to provide input to re-use the deployment for multiple projects, customers or purposes.

² <https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-best-practices/resource-naming>

Create the resource group

Like every Azure resource deployment, we start with creating a **Resource Group** to place all the Azure resources in.

```
resource "azurerm_resource_group" "rg" {
  name      = local.rgName
  location  = var.scaffold_location

  lifecycle {
    ignore_changes = [
      tags
    ]
  }
}
```

Create Dev Center resource and a Dev Center Project

The basis of the Managed DevOps Pools is the **Dev Center** resource, along with a **DevCenter Project**.

As described briefly earlier, the DevCenter is a collection of projects with similar settings. It can be used to supply catalogs with Infrastructure as Code templates, which are available for all projects in the DevCenter, as well as creating development environments for development teams to use.

A DevCenter Project is contained part that can be made available to specific teams and resources, i.e. Dev Boxes, Deployment Environments or Managed DevOps Pools.

```
resource "azurerm_dev_center" "devcenter" {
  name            = local.devCenterName
  location        = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
}
```

The DevCenter name cannot be longer than 26 characters, since we created the variables with validations, we should not reach this number. However, when you update the sample code and update the naming convention or variables, be aware you might reach this naming length restriction.

```
resource "azurerm_dev_center_project" "devcenter_project" {
  name            = local.devCenterProjectName
  dev_center_id   = azurerm_dev_center.devcenter.id
  location        = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
}
```

Create the virtual network and subnet

Now that we have the Dev Center set up, we can move forward to create the **virtual network** and **subnet** to be used by the Managed DevOps Pool to allow for private networking.

```
resource "azurerm_virtual_network" "vnet" {
  name            = local.vnetName
  location        = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  address_space   = [var.vnet_devpool_ip_range]
  dns_servers     = var.vnet_dns_servers
}
```

```
# Optionally peer the virtual network to the Virtual Hub
resource "azurerm_virtual_hub_connection" "agents" {
  depends_on = [azurerm_virtual_network.vnet]
  count      = var.virtual_hub_id != null ? 1 : 0
```

```
  name            = "conn-${local.vnetName}"
  internet_security_enabled = true
  virtual_hub_id   = var.virtual_hub_id
  remote_virtual_network_id = azurerm_virtual_network.vnet.id
}
```

Since I am always using a Cloud Platform, I want to link my virtual network to the centralized hub. You can choose to not use this resource, by not providing a value (or providing the value **null**) to the variable **virtual_hub_id**, if you do not want to use any connection with a Virtual Hub. If you want to know more about what a hub is, please have a look at the [Microsoft Learn pages about hub-and-spokes as part of the Cloud Adoption Framework³].

```
resource "azapi_resource" "snet" {
  depends_on = [azurerm_virtual_network.vnet]

  name      = local.snetName
  type      = "Microsoft.Network/virtualNetworks/subnets@2023-11-01"
  parent_id = azurerm_virtual_network.vnet.id

  body = jsonencode({
    properties = {
      addressPrefix = var.vnet_devpool_ip_range
      delegations = [
        {
          name = "Microsoft.DevOpsInfrastructure/pools"
          properties = {
            serviceName = "Microsoft.DevOpsInfrastructure/pools"
          }
        }
      ]
    }
  })
}
```

Next we use the earlier referred to **AzAPI** resource in order to create a delegated subnet for the Managed DevOps pool. We need this AzAPI resource because the **AzureRM** provider does not provide a known terraform configuration for the **Microsoft.DevOpsInfrastructure/pools** delegation service name. Using the **AzAPI** resource we can pass our delegation information conveniently.

Creating the Managed DevOps Pools

Now we have the fundamental resources we are going to create the Managed DevOps Pool, again using **AzAPI** provider.

```
resource "azapi_resource" "pool" {
  name      = local.poolName
  type      = "microsoft.devopsinfrastructure/pools@2024-04-04-preview"
  location  = azurerm_resource_group.rg.location
  parent_id = azurerm_resource_group.rg.id
}
```

³ <https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-best-practices/hub-spoke-network-topology>

```

body = jsonencode({
  properties = {
    organizationProfile = {
      organizations = [
        {
          projects      = var.devops_projects
          url            = var.devops_organization_url
          parallelism    = var.agent_maximumConcurrency
        }
      ]

      kind = "AzureDevOps" # Currently only Azure-
                           # DevOps is supported
      permissionProfile = {
        kind = "CreatorOnly" # Can also be set to
                           # "Inherit" or "SpecificAccounts"
        # If you want to use specific accounts, you
        # can add them here using the users and groups
        # properties
        # users = [
        #   "Patrick.deKruijf@xebia.com"
        # ]
        # groups = []
      }
    }

    devCenterProjectResourceId = azurerms_dev_center_
    project.devcenter_project.id

    maximumConcurrency = var.agent_maximumConcurrency

    agentProfile = {
      kind = "Stateless" # I would recommend setting
                        # this to "Stateless", since this ensures a fresh
                        # agent is used for each job.
      # kind = "Stateful"
      # maxAgentLifetime = "7.00:00:00" # Property
      # is required when set to "Stateful"

      # If you do not want to turn off scaling, remove
      # the complete resourcePredictionsProfile block
      # There is also a "Manual" option, which allows
      # you to set the minimum and maximum number of
      # agents based on a schedule.
      resourcePredictionsProfile = {
        predictionPreference = "MostCostEffective"
        # There are 5 options, ranging from "Most-
        # CostEffective" to "MostPerformance"
        kind = "Automatic"
        # Can also be set to Manual or
      }
    }

    fabricProfile = {
      sku = {
        name = "Standard_D2ads_v5"
      }

      images = [
        {
          aliases      = ["ubuntu-22.04"]
          buffer        = "*"
          wellKnownImageName = "ubuntu-22.04/latest"
        },
        # You can add more images if needed, also
        # referencing resource IDs for images
        # {

```

```

        # resourceId = "/Subscriptions/5ab24a52-
        # 44e0-4bdf-a879-cc38371a4403/Providers/
        # Microsoft.Compute/Locations/westeurope/
        # Publishers/canonical/ArtifactTypes/VMImage/
        # Offers/0001-com-ubuntu-server-focal/Skus/
        # 20_04-lts-gen2/versions/latest",
        # buffer      = "*"
        # }
      ]

      osProfile = {
        # Not much to configure here just yet, but
        # Microsoft is working on adding Key Vault
        # support too
        secretsManagementSettings = {
          observedCertificates = [],
          keyExportable        = false
        },
        logonType = "Service" # Can also be set to
        "Interactive"
      },

      # If you want to use an isolated network,
      # remove the complete networkProfile block
      networkProfile = {
        subnetId = azapi_resource.snet.id
      }

      storageProfile = {
        osDiskStorageAccountType = "Premium",
        # Standard, StandardSSD, Premium
        dataDisks = [
          # Create additional data disks if needed
          # {
          #   diskSizeGiB      = 100
          #   caching          = "ReadWrite"
          #   storageAccountType = "StandardSSD_LRS"
          #   driveLetter      = "Z"
          # }
        ]
      },

      kind = "Vmss" # Currently only "Vmss" is supported
    }
  }
})
}

```

For each of the properties a section is created below to explain what each setting expects, what the options are and when you should use a certain option.

Available properties

organizationProfile

```

organizationProfile = {
  organizations = [
    {
      projects      = var.devops_projects # This field
      accepts an Array[] of projects that should be able
      to use the Managed DevOps Pool
      url            = var.devops_organization_url # This
      should be the URL of a Azure DevOps organization
      (i.e. https://dev.azure.com/{organizationName})
      parallelism    = var.agent_maximumConcurrency # This
      setting sets the maximum amount of concurrent agents
      that can be used in parallel
    }
  ]
}

```



- [1] Code Repository <https://github.com/patrick-de-kruif/managed-devops-pools>
- [2] Microsoft Learn <https://learn.microsoft.com/en-us/azure/devops/managed-devops-pools/?view=azure-devops>
- [3] Origin Story <https://devblogs.microsoft.com/engineering-at-microsoft/managed-devops-pools-the-origin-story/>
- [4] Public preview announcement <https://devblogs.microsoft.com/devops/managed-devops-pools/>

```
kind = "AzureDevOps" # Currently only AzureDevOps is
supported, hopefully we can see GitHub here soon too
permissionProfile = {
  kind = "CreatorOnly" # Can also be set to "Inherit"
  or "SpecificAccounts"
  users = [ # If you want to use specific accounts,
  you can add them here using the users and groups
  properties
    "Patrick.deKruifj@xebia.com"
  ]
  groups = [] # If you want to use specific accounts,
  you can add them here using the users and groups
  properties
}
}
```

devCenterProjectResourceId

```
devCenterProjectResourceId = azurerm_dev_center_
project.devcenter_project.id
```

maximumConcurrency

```
maximumConcurrency = var.agent_maximumConcurrency
```

agentProfile

```
agentProfile = {
  kind = "Stateless" # You can use either "Stateful"
  or "Stateless". I would recommend setting this to
  "Stateless", since this ensures a fresh agent is used
  for each job.
  # maxAgentLifetime = "7.00:00:00" # Property is
  required when the kind property is set to "Stateful"

  # If you do not want to turn off scaling, remove the
  complete resourcePredictionsProfile block
  # There is also a "Manual" option, which allows you to
  set the minimum and maximum number of agents based
  on a schedule.
  resourcePredictionsProfile = {
    predictionPreference = "MostCostEffective"
    # There are 5 options, ranging from "MostCost-
    Effective" to "MostPerformance"
    kind = "Automatic"
    # Can also be set to Manual
  }
}
```

```
fabricProfile = {
  sku = {
    name = "Standard_D2ads_v5" # Make sure this SKU is
    allowed based on the Managed DevOps Pool quotas
  }
}
```

```
images = [
  {
    aliases = ["ubuntu-22.04"]
    buffer = "*"
    wellKnownImageName = "ubuntu-22.04/latest"
  },
  # You can add more images if needed, also
  referencing resource IDs for images
  # {

```

```
# resourceId = "/Subscriptions/5ab24a52-44e0-4b
df-a879-cc38371a4403/Providers/Microsoft.Compute/
Locations/westeurope/Publishers/canonical/Artifact-
Types/VMImage/Offers/0001-com-ubuntu-server-focal/
Skus/20_04-lts-gen2/versions/latest",
# buffer = "*"
# }
]

osProfile = {
  # Not much to configure here just yet, but Microsoft
  is working on adding Key Vault support too
  secretsManagementSettings = {
    observedCertificates = [],
    keyExportable = false
  },
  logonType = "Service" # Can also be set to
  "Interactive"
},

# If you want to use an isolated network, remove the
complete networkProfile block
networkProfile = {
  subnetId = azapi_resource.snet.id # This is the
  resource ID of the virtual network you want to have
  the Managed DevOps Pool connect to
}

storageProfile = {
  osDiskStorageAccountType = "Premium", # Standard,
  StandardSSD, Premium
  dataDisks = [
    # Create additional data disks if needed
    {
      diskSizeGiB = 100
      caching = "ReadWrite"
      storageAccountType = "StandardSSD_LRS"
      driveLetter = "Z"
    }
  ]
},

kind = "Vmss" # Currently only "Vmss" is supported
}
```

Required traffic rules (firewall or network security groups)

In order to all the resources actually work, you will need to allow traffic to specific domains. So these domains needs to be allowed from a network perspective to make the Managed DevOps Pool functional.

Endpoints that the Managed DevOps Pool service depends on:

- **.prod.manageddevops.microsoft.com** - Managed DevOps Pools endpoint
- **rmprodbuilds.azureedge.net** - Worker binaries
- **vstsagentpackage.azureedge.net** - Azure DevOps agent CDN location
- ***.queue.core.windows.net** - Worker queue for communicating with Managed DevOps Pools service
- **server.pipe.aria.microsoft.com** - Common client side telemetry solution (and used by the Agent Pool Validation extension among others)

- **azure.archive.ubuntu.com** – Provisioning Linux machines
– this is *****HTTP*****, not HTTPS
 - **www.microsoft.com** – Provisioning Linux machines
 - **packages.microsoft.com** – Provisioning Linux machines
 - **ppa.launchpad.net** – Provisioning Ubuntu machines
 - **dl.fedoraproject.org** – Provisioning certain Linux distros
- Needed by Azure DevOps agent:
- **dev.azure.com**
 - ***.services.visualstudio.com**
 - ***.vsblob.visualstudio.com**
 - ***.vssps.visualstudio.com**
 - ***.visualstudio.com** – These entries are the minimum domains required. If you have any issues, see Azure DevOps allowlist^[4] for the full list of domains required.

For more information regarding the outbound traffic, please see this Microsoft learn page^[5].

Start the deployment

To start the deployment, we will need to provide the deployment with the values for the variables required. See an example file below:

```
scaffold_location           = "westeurope"
scaffold_environment        = "production"
scaffold_environment_short_name = "prod"
scaffold_location_short_name = "weu"
scaffold_company_short_name  = "{insert-your-company-short-name}"

virtual_hub_id              = "/subscriptions/{insert-your-hub-subscription-id}/resourceGroups/{insert-your-hub-resource-group-name}/providers/Microsoft.Network/virtualHubs/{insert-your-virtual-hub-name}"
vnet_devpool_ip_range      = "{insert-your-ip-range}"
vnet_dns_servers            = ["{insert-your-dns-server-ip}"]
agent_maximumConcurrency   = 2 # This is the maximum number of agents that can run concurrently, keep in mind the SKU quota you have on your subscription
devops_organization_url     = "https://dev.azure.com/{insert-your-organization-name}"
devops_projects             = ["{insert-your-project-name}", "{insert another project name}"]
```

Since we're using Terraform, we can simply run the following commands:

```
terraform init # This will initialize the backend settings and install the required providers
terraform validate # This will check if all
terraform plan --var-file=test.tfvars # This will execute a 'dry-run' to see what would be created, modified or removed, using the tfvars-file referenced
terraform apply --var-file=test.tfvars # This will firstly do another dry-run, asking a 'yes' to continue with the actual deployment, using the tfvars-file referenced
```

i NOTE

That this will create a local Terraform state file, which is fine for my demo purposes. When you are using this in a production environment, please update the state backend accordingly.

For the explanation of the deployment and required steps, we use the commands directly into a terminal. We prefer and recommend to use Azure Pipelines or GitHub Actions to deploy Infrastructure as Code.

Cost of using DevOps Pools

Managed DevOps Pools pricing is determined by the cost of the Azure services your pool uses, like compute, storage, and data egress, combined with the standard Azure DevOps Services pricing for self-hosted agents.

- **Azure Services** – The Managed DevOps Pool uses Azure resources to supply the functionality. These resources will be billed to your Azure subscription. During public preview there are no extra costs for the Managed DevOps Pools resource itself
- **Azure DevOps Services** – The cost for the DevOps Services are tied to the costs of self-hosted agents, this means that you will have to pay for parallel jobs. The first parallel job is free, then it is set to \$15 per additional parallel job.

Please note that parallel jobs are shared between all pipelines and pool in your organization.

Conclusion

In my opinion using Managed DevOps Pools is definitely worth it. You get the ease of mind, because it is a PaaS offering. It directly integrates into your private network within your Azure tenant, allowing for better and safer connections. And with the code repository, you will get a quick starter template to be able to use it. A Managed DevOps Pool is only available for the teams that have access to it and you can create multiple pools with different settings, i.e. a CPU-intensive pool and a Memory-intensive pool for different teams.

Microsoft is also considering adding support for container-based agents to improve on-demand spin up times. I am excited to see the improvements in the future!

i Additional information

Origin story

In case you want to read more about the origin story, please read the story by Suraj Gupta and Elize Tarasila^[6].

⁴ <https://learn.microsoft.com/en-us/azure/devops/organizations/security/allow-list-ip-url?view=azure-devops&tabs=IP-V4>

⁵ <https://learn.microsoft.com/en-us/azure/devops/managed-devops-pools/configure-networking?view=azure-devops&tabs=azure-portal#restricting-outbound-connectivity>

⁶ <https://devblogs.microsoft.com/engineering-at-microsoft/managed-devops-pools-the-origin-story>

Kubernetes Network Policy



Zero Trust, Network segmentation

By default Kubernetes networking is not secure. All network traffic is open and all containers are accessible over the network. One container can connect to every other container in the cluster freely. Every port is open for ingress and egress traffic. This is fine if you trust everybody in your environment but not so fine if your environment is shared with multiple teams across your organization. Also when you are running workload that is maintained by external parties or from public and open sources you should be extra careful and make sure it can only access the resources it needs to access.

Author Robert de Veen

To protect your workload from unauthorized access from "malicious" attackers you can use Kubernetes Network Policies to close down network access to your workload. You can see it like a firewall or compare it to an Azure Network Security Group on a virtual network. According to the Zero Trust principle "Assume breach", you should be prepared for attackers in your clusters network. Assuming that an attacker has access to run some malicious workload in your cluster, it should not be able to connect to other resources in your network. You should protect your workload even from access by the neighbor running workload. Your network should be segmented into many small segments. Each segment should have its own ingress and egress ports being opened. Traffic not allowed into or from the segment should not be permitted, you only want traffic that you explicitly specify and allow.

Network Policy structure

Without a network policy every address and every port is open. A policy can specify if you would like to block all 'ingress' or all 'egress' or both by setting this in the policy type. The next step is to open specific ports for specific workloads so that they can communicate with each other. You can do this by specifying the 'podSelector' and the 'port' in the policy. The 'podSelector' is a label selector to specify the pods that the policy applies to. The 'port' is the port number that is being opened. With the 'namespaceSelector' you can also specify the namespace of the traffic origin or destination.

The policy is deployed into a Kubernetes namespace. This means that the policy is applied to every workload in that namespace. If you create an empty policy of type 'ingress', that will mean that every ingress traffic is then being blocked. You have to specify which traffic you want to allow. The same principle applies to an 'egress' policy. Network policies do not conflict, they are additive. If you create multiple policies they all apply, you cannot close a port again that is already opened by another policy.





How to apply a network policy

When you want to apply a network policy in Kubernetes, you can use the **kubectl apply --filename network-policy.yaml** command. This will apply the network policy to the active namespace or specify the namespace in the command **kubectl apply --filename network-policy.yaml --namespace my-namespace** or set it in the definition. To verify if the policy is applied correctly, you can use the **kubectl describe networkpolicy <name-network-policy>** command.

The following example configuration will block all incoming and outgoing traffic. This will ensure that pods from other namespaces will not be allowed to open a network connection to any workload in the namespace. The workload in the namespace can't open any network connection to resources outside of the namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

Because the **podSelector** is an empty object, the policy will apply to all the pods in the namespace. The **policyTypes** specifies that both ingress and egress traffic is blocked.

Example solution

Lets take the following simplified example solution. We have an ingress controller specified in the namespace 'ingress-namespace'. Ingress traffic from the Internet is allowed and egress traffic to the Frontend pod is allowed. The Frontend pod is allowed to make a connection to the Backend pod. The Backend pod only allowed traffic from the Frontend pod. The Frontend and Backend pod are both in the 'app-namespace' Other connections are not allowed and should be blocked by the network policy.

```
flowchart LR
    Browser[Browser 🌐] -- ".Port: 443.->" --> Ingress
    subgraph Kubernetes Cluster
        subgraph ingress-namespace
            Ingress[Ingress controller]
        end
        subgraph app-namespace
            Ingress -- ".Port: 8443.->" --> FrontEnd
            FrontEnd[Frontend] -- ".Port 8080.->" --> API[Backend]
        end
    end
```

We create a network policy for the 'ingress-namespace' to allow the traffic. With the podSelector we specify that the policy is only applied to the pods with the label **app: ingress-controller**. Other pods in this namespace are still being blocked. You can specify specific pods by labels or set **podSelector: {}** to apply this policy to all pods in the namespace.

In the ingress section we specify from which IP address block incoming traffic is allowed, in this case every IP address in the world by using the 0.0.0.0/0 range. In the ports section we specify the port 443 to allow HTTPS traffic only.

For the egress the namespaceSelector and the podSelector are used to specify the frontend pod by label **app=frontend** in the **app-namespace**. The namespaceSelector only works on labels, and not on the name of the namespace. Kubernetes automatically applies a label **kubernetes.io/metadata.name** when you create a new namespace, you can use that label or add your own label to the namespace. So now the only outgoing traffic is allowed to the frontend pod in the app-namespace on port 8443.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-namespace
  namespace: ingress-namespace
spec:
  podSelector:
    matchLabels:
      app: ingress-controller
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 0.0.0.0/0
  ports:
    - port: 443
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: app-namespace
        podSelector:
            matchLabels:
              app: frontend
    ports:
      - port: 8443
```

For the 'app-namespace' we create a network policy to allow the incoming traffic. This policy will apply only to pods with label **app=frontend**. It allows ingress traffic from the ingress controller pods in the ingress namespace. Only port 8443 is opened.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-frontend
  namespace: app-namespace
spec:
  podSelector:
    matchLabels:
      app: frontend
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: ingress-namespace
        - podSelector:
            matchLabels:
              app: ingress-controller
  ports:
    - port: 8443
```

The following policy allows egress traffic from the frontend to the backend pod. This doesn't automatically allow the incoming traffic on the backend pod, this should be specified in another policy.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-frontend
  namespace: app-namespace
spec:
  podSelector:
    matchLabels:
      app: frontend
  policyTypes:
    - Egress
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: backend
  ports:
    - port: 8080
```

And the policy to allow the incoming traffic on the backend pod. This will block all other incoming traffic so no other pods can connect to the backend pod directly.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-backend
  namespace: app-namespace
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
```

```
ingress:
  - from:
      - podSelector:
          matchLabels:
            app: frontend
    ports:
      - port: 8080
```

With these policies applied to the Kubernetes cluster, we have blocked all unwanted network traffic. We can fine-grained allowing access from specific workload to specific workload over specified ports. All other traffic is being blocked. This will secure your workload against any unauthorized and unwanted access over the network. Happy securing your Kubernetes cluster!

Lesions learned 1: Multiple workload in same namespace

If you have multiple workloads running in the same namespace please pay attention. When deploying a policy, the definition is applied to all the workload in that namespace. So if you apply a policy to disable ingress it will be disabled for all the workload in that namespace, unless you explicitly allowing it for that workload again.

Lesions learned 2: DNS Requests

When you apply the network policy to block all egress traffic, the DNS request are also being blocked. Even if you allow your pod to make a connection to for example <https://www.xebia.com>, the pod doesn't know the IP address of that website. To get the IP address of the website the pod asked the internal DNS server running in the cluster for the IP address. These request are using port 53. So if you block all egress traffic, you also block the DNS request. To allow the DNS request you have to open the port in the network policy.

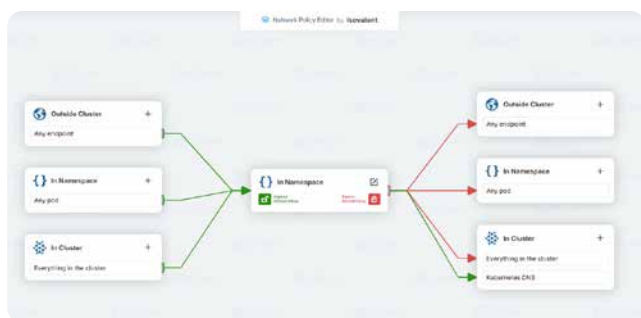
Enable DNS

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: enable-dns-network-policy
spec:
  policyTypes:
    - Egress
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: kube-system
          podSelector:
            matchLabels:
              k8s-app: kube-dns
    ports:
      - port: 53
        protocol: UDP
```



TIP**Network Policy Editor**

To help with building your first network policy, you can use the **Network Policy Editor for Kubernetes** at <https://app.networkpolicy.io/>. This app will visualize the policy configurations and allows you to better understand the possible rules you can specify. It will visualize the rules you have specified and give you the correct yaml definition. A helpful tool to understand the different settings of the complex network policy definition.

**TIP****Kubernetes Network Policy Recipes**

On <https://github.com/ahmetb/kubernetes-network-policy-recipes> you can find a lot of examples of network policies. You can use these examples to build your own policy and use them as a starting point for your own policy.

NOTE**Supported Network Policy Managers for Azure Kubernetes Services (AKS)**

This article describes network policies for Kubernetes. The network policy engine is responsible for enforcing the network policies. Azure Kubernetes Services (AKS) supports multiple network policy engines.

• None

Without a network policy engine, the applied network policy is not being used. If you deploy a policy, but no policy engine, nothing will happen and all traffic is still open.

• Azure Network Policy Manager

The Azure Network Policy Manager is a Microsoft managed network policy engine. It used the Kubernetes network policy definition to enforce the network policies. It is a simple network policy engine and doesn't support all the features of the other engines but a good starting point for simple network policies.

• Cilium

Microsoft recommend <https://learn.microsoft.com/en-us/azure/aks/use-network-policies#network-policy-options-in-aks> Cilium as their preferred network policy engine for AKS. It is the most feature-rich engine and is actively supported and maintained by Microsoft. It has support for filtering on FQDN or HTTP methods. A global network policies (CiliumClusterwideNetworkPolicy) to specify non-namespaces and so cluster-scope policies is also possible. For network observability, it provides Hubble UI to visualize the network traffic.

• Calico

The other out-of-the-box solution for AKS is Calico. It is build by Tigera and has support for Linux and Windows Server nodes. The way policies are specified is more advanced than the Kubernetes network policy definition. It has global network policies (GlobalNetworkPolicy) to specify non-namespaces and so cluster-scope policies. But the basic version doesn't support filtering on FQDN or HTTP methods.

• BYOCNI

BYOCNI stands for **Bring Your Own Container Network Interface**. It allows you to deploy an AKS cluster with no CNI plugin preinstalled. You can install any third-party CNI plugin such as Cilium, Flannel and Weave and use their implementation of network policies. The downside is you have to manage the CNI plugin and the policy manager yourself.

How to build a maintainable and highly available Landing Zone

This is how we design, build, deploy, and maintain a Multi-Region Azure Landing Zone. We do this while delivering the Landing Zone like a Platform as a Product. In this article, you will find some tips and tricks to make the setup of your landing zone a bit easier.

Authors Jelmer de Jong and Arjan van Bekkum

What is an Azure Landing Zone?

An Azure Landing Zone is a pre-configured environment within Microsoft Azure designed to provide a secure and scalable foundation for your cloud workloads. Its architecture is modular and scalable, allowing you to apply configurations and controls consistently across your resources. It uses subscriptions to isolate and scale application and platform resources¹.

Every Landing Zone we build is based on the **Cloud Adoption Framework (CAF)**. Microsoft created the cloud adoption framework to bring cloud adoption for Azure customers. It helps to achieve the best business outcome for cloud adoption. The CAF is a set of best practices for setting up Azure Infrastructure. It follows key design principles across eight design areas to enable application migration, modernization, and innovation at scale.

There are two main types of landing zones:

1. **Platform Landing Zones:** These provide shared services (like identity, connectivity, and management) to applications in workload landing zones. Central teams manage them to improve operational efficiency. The platform Landing Zone is called a Hub.
2. **Workload Landing Zones:** These are environments deployed for the workloads. The workload landing zone is called a spoke.

Azure Resource Structure

Azure provides four levels of management: management groups, subscriptions, resource groups, and resources. The following diagram shows the relationship between these levels.

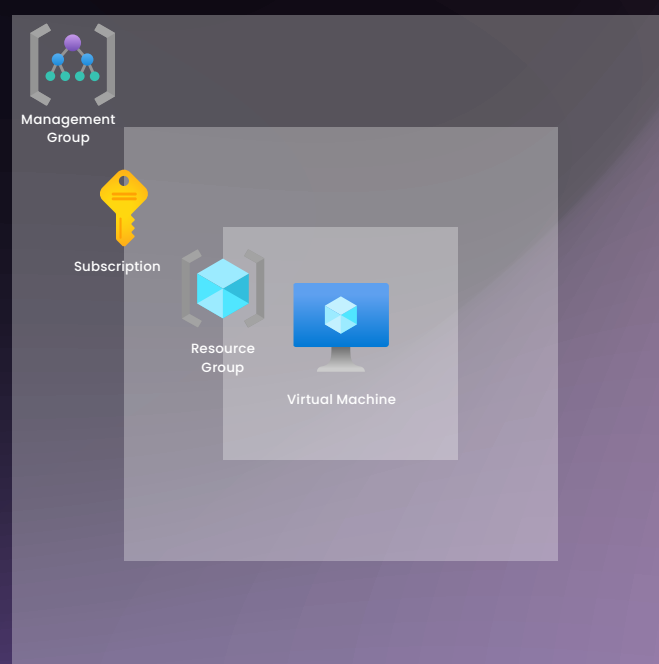


Figure 1: Azure Structure

¹ <https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/>

Management groups help you manage access, policy, and compliance for multiple subscriptions. All subscriptions in a management group automatically inherit these settings applied to the management group.

Subscriptions logically associate user accounts with the resources. Organizations can use management groups, subscriptions, and resource groups to manage costs and the resources users, teams, and projects create. Each subscription has limits or quotas on the number of resources that can be created and used.

Resource groups are logical containers in which you can deploy and manage Azure resources such as virtual machines, web apps, databases, and storage accounts.

Resources are instances of services that you can create in a resource group, such as virtual machines, storage, and SQL databases.

Landing Zone Design Areas

We have defined nine core design areas when designing, building, and maintaining an Azure Landing Zone.

- Platform Landing Zone / Hub Design
- Infrastructure as Code
- Zero Trust Architecture
- Multi-Region
- Monitoring
- Alerting
- Networking
- Policies and Guardrails
- Architectural Decisions Records (ADR)

Platform Landing Zone Design

The Platform Landing Zones or Hub consists of three parts (identity, connectivity, and management) and provides shared services for all application workloads. Most of the hub resources are deployed in every region to ensure that if one region fails, it will not affect the whole landing zone.

The **management part** of the Hub contains all Azure resources for managing the Azure Landing Zone. For example, there are central log analytics workspaces for all central log metrics and (audit) or resources. The management part also contains a central automation account.

The **connectivity part** of the hub contains all Azure resources needed for connectivity. Here is where we find the Virtual WAN and all the Virtual Hubs, as well as the firewalls connected to those hubs and, of course, the express route circuits and Peer to Site and Site to Site Gateways. The DNS private zones are also deployed in the connectivity part. They are the most crucial part of the Landing Zone and make the traffic secure and operational.

The **identity part** is where your Entra ID or Domain Controllers are located.

Infrastructure As Code

Infrastructure as Code or IaC is one of the core principles of a Landing Zone. We deploy every Azure resource using infrastructure as code. We do this for the following reasons:

Resource Consistency: Imagine deploying fifteen firewall rules across two regions, West Europe (WE) and North Europe (NE). Manually applying these rules increases the risk of human error, potentially leading to inconsistent firewall configurations between the regions. Using Infrastructure as Code (IaC) ensures that the rules are consistently deployed across both regions.

Scalability or ease of deployment: Consider deploying five virtual machines. Which approach is more scalable and manageable, manually setting up each VM or using Infrastructure as Code (IaC) to deploy all five? We will always choose the latter. With IaC, you can define the VM once and deploy it multiple times. By using IaC, you ensure each instance is identical, and you reduce the risk of errors.

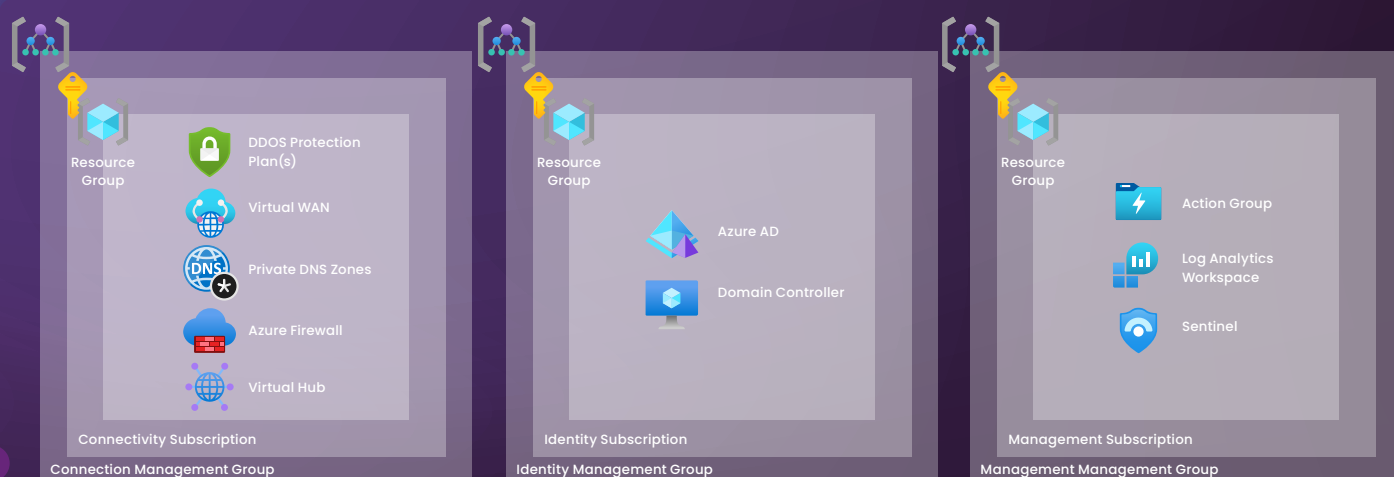


Figure 2: CAF Structure

Version Control: We place the IaC configuration in a Version Control system like GIT. We do this so we can keep track of changes that are being made and also because we want to be able to review each other's work, especially when we are deploying or changing something in production. Version control also allows you to test changes on a separate environment like Sandbox before applying them to production. We achieve all these benefits by combining Version Control and IaC.

Terraform

We build almost every Azure Landing Zone using Terraform as the IaC framework. We won't go into detail why, but these are the main reasons:

- Easy to understand
- Can automate beyond cloud resources
- Findable talent in the market

However, it is important to note that we use Terraform in a very opinionated way. For example, we split Terraform code into Modules, Templates, and Solutions. This opinionated approach ensures that each piece of the infrastructure is modular and reusable. By splitting Terraform into Modules, Templates, and Solutions, we maintain a clean separation of concerns, improve scalability, and streamline upgrades.

Modules: Modules are predefined, opinionated resource definitions. An example could be a Virtual Machine (VM). However, not every resource definition needs to be a module. If your module doesn't provide meaningful abstraction or opinionation, it shouldn't exist. Avoid creating thin wrappers, as they add unnecessary complexity.

Additionally, modules should not consume other modules, known as nested modules, because keeping the module structure flat simplifies code management and reduces the complexity of making changes.

Every module is a separate repository and has an example. We use this example to test the module or like a quickstart for those looking to consume it. Whenever we create or update a module, our pipeline releases a new version. When consuming modules in templates or solutions, a version number is supplied. This number corresponds to one of the release modules. This allows us to manage and control when and how we upgrade the modules we use. More details on this will be covered in the Layered deployment section.

Templates: Templates are a combination of modules and resource definitions. They represent frequently used deployment patterns. An example of this is a Virtual Machine with backup agents, and more if needed. Templates are like modules; they should not consume templates again, so making code changes stays simple.

Templates are just like modules placed inside separate repositories and versioned when created or modified.

Solutions: Solutions represent deployable Azure Resources for a specific project.

They do this by consuming either modules or templates or calling resource definitions directly.

What is a thin wrapper in Terraform? A thin wrapper is a module that adds little to no value. It allows you to pass all settings as variables without adding any internal logic or abstraction. Essentially, it only lightly wraps a resource definition, which is why it's called a "thin wrapper."

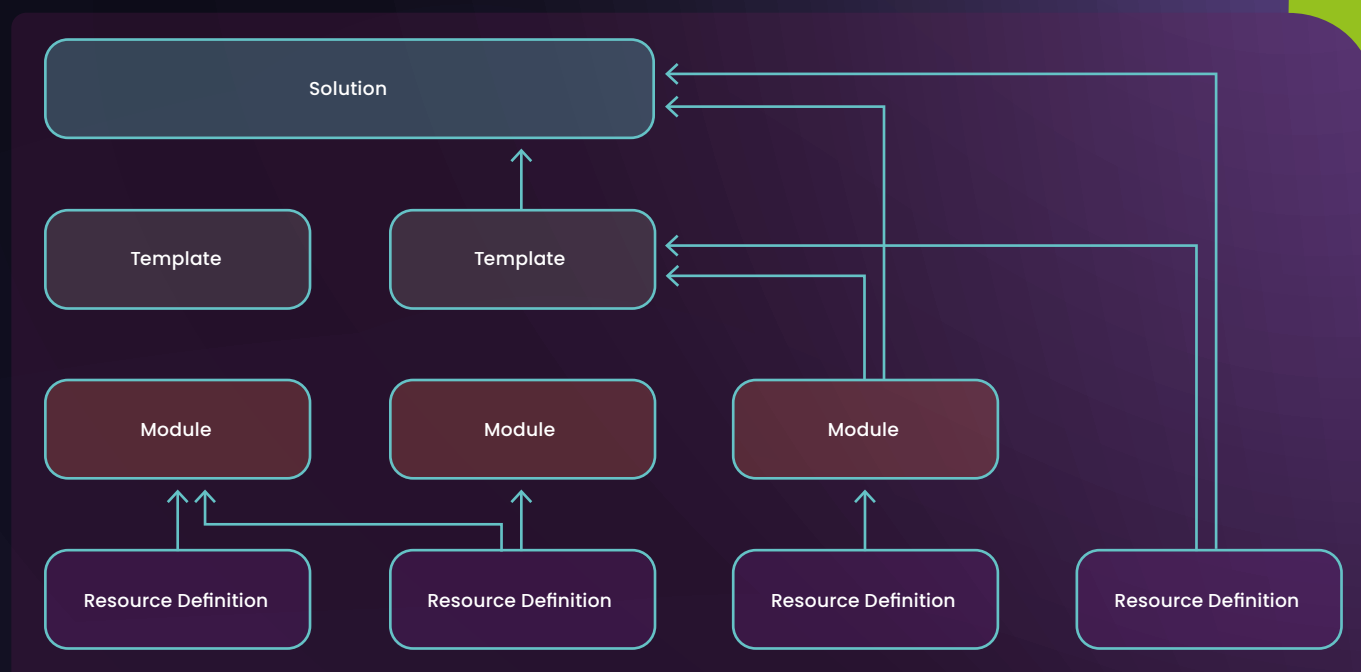


Figure 3: Terraform setup

Combining these three groups will create a structure like the image below. You can see that modules consume resource definitions. The template can consume modules and resource definitions. And lastly, solutions can consist of resource definitions, modules, and templates.

Terraform State

Terraform uses the state file (terraform.tfstate) to keep track of the resources it manages. It acts as a source of truth for the infrastructure, storing information about the current state of the managed infrastructure. The state file ensures that Terraform has an accurate and consistent view of the infrastructure, essential for making correct updates and changes. The state file helps Terraform understand resource dependencies, ensuring that resources are created, updated, or destroyed in the correct order. Every deployment Terraform locks the state file, allowing only the planned changes to be applied and preventing two deployments at the same time. The state file is stored remotely in an Azure Storage Account; this way, the state file can be used by all the team members, enabling collaborative workflows.

However, the state file can contain sensitive information (e.g., resource IDs, secrets). Proper security measures, like encryption and limited access to the storage account, must be taken to protect it.

However, the state file can contain sensitive information (e.g., resource IDs, secrets). Proper security measures, like encryption and limited access to the storage account, must be taken to protect it.

```
terraform {
  backend "azurerm" {
    container_name      = "tfstate"
    key                 = "vm.terraform.tfstate"
    resource_group_name = "rg-state-prod-we-001"
    storage_account_name = "mystorageaccount"
    subscription_id     = "12345678-aa99-bb88-55cc-098765432123"
  }

  required_version = "~> 1.1, <= 1.8.1"

  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "3.92.0"
    }
  }
}
```

Zero Trust Architecture

This principle advocates a security approach where services within the cloud don't automatically trust users or other services, regardless of their network or location. Instead, it emphasizes continuous verification and authentication, granting access on a need-to-know basis.

Organizations implementing Zero Trust Architecture fortify their security posture, effectively mitigating unauthorized access and data breach risks. Key aspects of this principle include:

Continuous Verification: The Zero Trust Architecture implemented in the Landing Zone ensures that every access request is continuously verified, regardless of the user's location or network. This involves multiple authentication and authorization layers, including user identity verification, device health checks, and contextual information such as location and behavior patterns.

Least Privilege Access: Access to resources within the Landing Zone follows the principle of least privilege. Users are granted the minimum necessary privileges to perform their assigned tasks, and access rights are continuously evaluated and adjusted based on changing requirements or roles. This minimizes the attack surface and mitigates the potential impact of compromised accounts.

Micro-Segmentation: The Landing Zone implements micro-segmentation to create security boundaries within the environment. By dividing the network into smaller segments, each with access controls, traffic can be effectively controlled and isolated. This approach limits lateral movement in the event of a security breach, reducing the impact on the overall system.

Management groups

Several default management groups are deployed for both types of the landing zone. The Platform landing zone is available under the "Azure Platform Services" management group, and each of the three parts of the landing zone has its management group.

A management group, "Landing Zone," is created for the workload landing zone. Below these groups, a management group per geographical region, such as "EMEA", is deployed. A management group per Azure region is deployed to provide maximum flexibility, like in West Europe. If needed, you can distinguish between production and nonproduction per Azure Region.

Subscriptions

Each part of the landing zone has its subscription. This goes for the three parts of the platform landing zone as well as the workload landing zone. Subscriptions are not shared between workloads, in line with the zero trust principles. With a landing zone in multiple regions each region will get its own subscription to host the workload. This segmentation also helps in case of regional disaster. Only the affected region will be unavailable, not the entire landing zone.

Of course there are also global resources like the Virtual WAN. Global resource will be deployed only once.



Terraform

These resources still require a subscription and a resource group. To deploy, simply select the subscription of your choice to deploy these resources.

Multi-region

When setting up a landing zone, it is essential to consider the number of regions needed. Do you have an application that requires 24x7 uptime? Do you have users around the globe? Are there special requirements for data storage? Are there compliance regulations? Answering these questions will give you more insight into the regions you need.

Next to the number of regions, it is also important to verify the resources you want to use. Not every Azure Region contains the same resources, so select a region where the needed resources are available. And last but not least, think about the capacity you need. A popular region is, for example, West Europe; it could happen that because of the high demand, capacity runs low, so you can maybe select a less popular region that also meets your needs.

Monitoring

Monitoring is an essential part of both landing zones. Without it, you cannot see what is happening in your landscape. All resources we deploy will also get diagnostic settings. For the workload landing zone, each workload gets its own log analytics workspace, which can be used to monitor the workload closely. Remember, we use isolation in the landing zone, so a shared log analytics workspace for multiple workloads will cause some issues with sharing data.

However, every region has a log analytics workspace in the platform landing zone. These log analytics workspaces are connected to Sentinel^[2], the SIEM tool in Azure that detects threats and suspicious activity. To ensure we can also use Sentinel for all the workloads, all the log analysis workspaces for the applications are connected to the central ones. This allows Sentinel to detect activity throughout the whole landing zone.

Alerting

Alerts are essential to monitoring; they actively trigger you when something happens. Creating alerts can be a lot of work. The question "What are we going to fire an alert on?" is common. You can use the Azure Monitor Baseline Alerts^[3] (AMBA) to kickstart implementing alerts in the landing zone. This GitHub repository contains a lot of policies that deploy alerts for a lot of resources in Azure.

The AMBA policies will trigger when a new resource is deployed, and alerts are automatically created. For example, for every express route circuit, you will get alerts to monitor the drop-off of packages. And when you connect those alerts to, for example, an action group that posts a message in a Slack channel or Teams chat, you have a good starting point. Of course, be a bit careful; the policies might deploy

alerts that trigger and notify too often, so you need to tweak and tune it until you are satisfied with when and how often the alerts trigger.

Networking

Hub and Spoke Network Architecture

The Hub and Spoke Network Architecture forms the backbone of the Landing Zone, facilitating a structured and scalable network design. The Landing Zone Hub is a central point of connectivity for both the spokes and the remote locations. In a Hub and Spoke architecture, all traffic leaving the spoke will go through the hub. Inside the Hub, all traffic is constantly monitored and verified. This way, only trusted traffic can leave and enter the spoke.

It is easy to compare the hub and spoke model with an airfield. The Hub is the central hall, and the spokes are the planes. To board a different plane, you must traverse the central hall to reach the other plane. You may go through customs but must show your boarding pass again when boarding the other aircraft.

Using a Hub and Spoke model makes it easier to manage core resources centrally, which is also cost-efficient. Using different spokes for different workloads improves security, as each spoke has its own access control set. By default, spokes cannot communicate with each other; we can allow this when needed. Lastly, this model keeps your Landing Zone scalable. The architecture can grow as more spokes are added to support different workloads or teams without changing the core structure.

Virtual WAN

If you have an Azure landing zone, you might need a connection between multiple regions and remote locations. To make the connection between all those resources more accessible to maintain, you can use a Virtual Wan. Inside that virtual wan, you need to create virtual hubs. It is best practice to create a virtual hub for the Azure region. Each network created in that region is connected (or peered) to the virtual hub. The Virtual WAN uses BGP to promote the routes of all the networks connected to the virtual hubs. The Virtual WAN uses this Routing Intent settings to propagate the routes to the connected virtual networks. Because all the traffic between spokes traverses the hub and all the networks are connected, traffic can flow between spokes without any problems.

Firewall

Firewalling plays a crucial role in ensuring network security in the landing zone. As part of the architectural design, Azure firewalls are deployed and connected to each virtual hub. This centralizes the firewalling capabilities and provides a consistent security posture across the environment.

² <https://learn.microsoft.com/en-us/azure/sentinel/overview?tabs=azure-portal>

³ <https://github.com/Azure/azure-monitor-baseline-alerts>

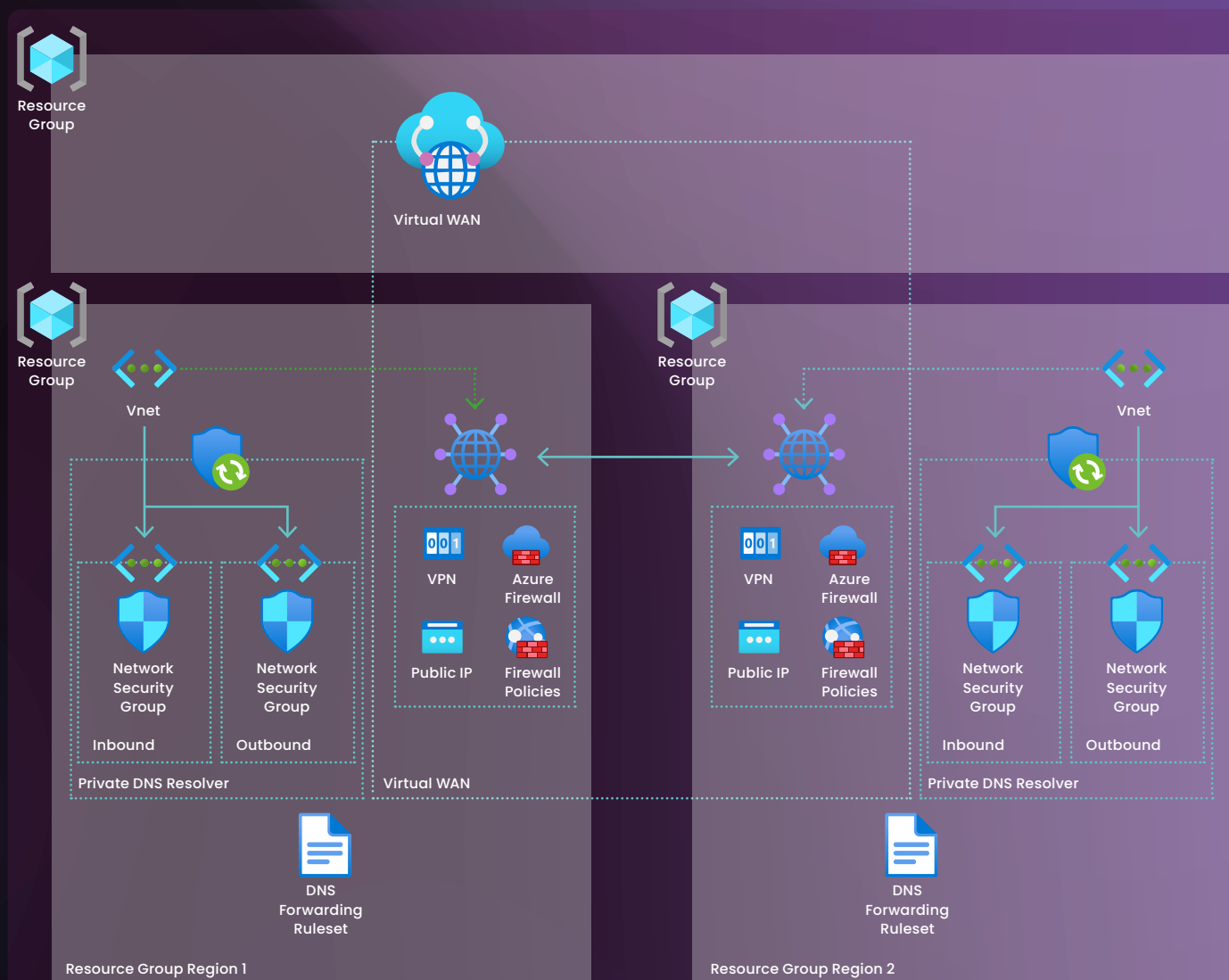


Figure 4: Virtual WAN

A premium Azure Firewall is deployed to provide maximum security features by default. This enables the use of advanced security features, such as:

- Intrusion detection and prevention system
- TLS Inspection
- URL filtering

By default, all traffic is blocked as part of the zero-trust architecture. Traffic that flows between workloads needs to be added to the firewall policies. Firewall rules are considered to be part of the infrastructure. Changes to these firewall rules must be made using Infrastructure as Code and the CI/CD pipelines. This approach ensures that changes to the firewall rules must be approved using the four-eyes principles and version control.

Routing

Besides network security, the firewall is also used as the next hop for all networks. The subnets inside the virtual network all have a routing table. The default rule in this routing table is that all traffic is routed to the firewall. This way, the traffic will traverse the virtual hub and thus virtual WAN.

Private endpoints

Private Endpoints in Azure are a network interface that connects you privately and securely to a service powered by Azure Private Link. Private Endpoints use a private IP address from your Virtual Network (VNet), effectively bringing the service into your VNet. This allows you to access the service without exposing it to the public Internet. Traffic will not use the Microsoft backbone or the public Internet to connect, lowering the latency and improve network security.

DNS

A crucial part of routing all the traffic between the workloads is DNS. In Azure, you can use the Azure Private DNS zones to set up your DNS. We already mentioned that the firewall is not only security-related; it can also be used as a DNS proxy. In order for the firewall to act as a DNS proxy, the DNS Servers for all the virtual networks need to be set to the regional firewall's (private) IP address. You need to enable the DNS settings and DNS proxy inside the firewall policy.

Private DNS Zones

All the private DNS zones are deployed in the connectivity part of the hub. These DNS zones must be globally available for all workloads in the landing zone. All the created private endpoints must have a DNS configuration that connects them to the Private DNS Zone. To make this work, a policy initiative is deployed. This initiative monitors the deployment of private endpoints and adds the DNS configuration to the correct private DNS zone.

```
resource "azurerm_management_group_policy_assignment"
  "deploy_private_dns_zones" {
    name                = "deploy_private_dns_zones"
    display_name        = "Configure Azure PaaS services
to use private DNS zones"
    policy_definition_id = "/providers/Microsoft.
Management/managementGroups/XebiaRoot/providers/
Microsoft.Authorization/policySetDefinitions/Deploy-
Private-DNS-Zones"
    management_group_id = azurerm_management_group.this.id
    # Both location and identity are needed for resource
    changes when the policy contains "modify" or
    "deployIfNotExists" effects.
    location = var.policy_location
    identity {
      type = "UserAssigned"
      identity_ids = [
        azurerm_user_assigned_identity.policy_assignment.id
      ]
    }
    parameters = <<PARAMETERS
    {
      # var.central_dns_zone contains the subscription id
      and the resource group name
      "azureFilePrivateDnsZoneId" : {
        "value": "/subscriptions/${split("/", var.central_
dns_zones)[0]}/resourceGroups/${split("/",
var.central_dns_zones)[1]}/providers/Microsoft.
Network/privateDnsZones/privatelink.afs.azure.net"
      },
      ... <long list of dns parameters> ..
    }
  }
PARAMETERS
}
```

DNS Forwarding Rulesets

By default, the DNS Setting on the firewall policy uses the Azure DNS to resolve the addresses. There is an option to provide a custom DNS server if Azure DNS is insufficient. One way to do this is to add private DNS Resolvers. A private DNS resolver has an inbound endpoint with an IP address; this address needs to be the custom DNS proxy forwarder on the firewall. If you, for example, have a custom domain controller for a specific domain, you also need to add a ruleset to the DNS Resolver. The traffic will be forwarded to the defined address if the rule matches the DNS requested.

Policies or Guardrails

Policies or guardrails are incorporated into the deployment process to ensure compliance and governance.

These guardrails check against predefined rules and guidelines, verifying that the deployed infrastructure and workloads align with security, compliance, and governance requirements.

Azure provides a set of policy initiatives, such as ISO 27001/27002, CIS (Center for Information Security), and NIST. In addition to the standard policies, customized policies like the Security Frameworks can be added to the policies in the Landing Zone.

The policies are set to the relevant management groups to ensure that the policies are active on all management groups and subscriptions (spokes and their resources).

All resources must follow the guardrail approach, which allows them a certain degree of freedom within defined boundaries. This approach balances flexibility and compliance by establishing guardrails consisting of policies and guidelines. These guardrails keep the landing zones in check, ensuring that environments can be customized while operating within the established policies.

Policies are written as code using Json and deployed with Terraform, enabling version control and automated deployment through CI/CD pipelines. This approach streamlines policy updates and ensures consistency across the landing zones. By treating policies as code, we can easily manage, update, and enforce them.

```
{
  "name": "DenyCreationTagWithCertainValues",
  "managementGroupId": null,
  "properties": {
    "displayName": "Custom - Allow resource creation if
tag value in allowed values",
    "policyType": "Custom",
    "mode": "All",
    "description": "Allows resource creation if the tag
is set to one of the following values.",
    "metadata": {
      "version": "1.1.0",
      "category": "Xebia Custom - General"
    },
    "parameters": {
      "tagName": {
        "type": "String",
        "metadata": {
          "displayName": "Tag Name",
          "description": "Name of the tag, such as
'Environment'"
        }
      },
      "tagValues": {
        "type": "Array",
        "metadata": {
          "displayName": "Tag Values",
          "description": "Values of the tag, such as
'PROD', 'TEST', 'QA', etc."
        }
      }
    }
  },
  "policyRule": {
    "if": {
      "allOf": [
        {
          "field": "type",
          "equals": "Microsoft.Resources/subscriptions/
resourceGroups"
        },

```

```

    {
      "not": {
        "field": "[concat('tags[' , parameters
          ('tagName'), ''])]",
        "in": "[parameters('tagValues')]"
      }
    }
  ],
  "then": {
    "effect": "Deny"
  }
}
}
}

```

Enterprise Scale

If you build a landing zone, you do not want to reinvent the wheel. This also applies to a lot of "best practice" policies. The Azure team maintains a repository with all the best practices for an enterprise's landing zone. Besides all kinds of examples for how to deploy the landing zone, it also contains a lot of policy definitions and policy initiatives^[4].

Assignments

As mentioned before, policies are assigned to a management group. Many policy definitions or initiatives contain input variables. These variables can be used to tweak and tune the policy the way you want it. You could, for example, set the "effect" of a policy to "Audit" instead of "Deny." This will still show if resources are not compliant but will not block a deployment. There are a lot of parameters you can set, so be sure to use them wisely.

```

resource "azurerm_management_group_policy_assignment"
"inherittagvalues" {
  name                = "inherittagvalues"
  display_name        = "Xebia - Configure - Inherit
    Tagging Values"
  policy_definition_id = "/providers/Microsoft.Management/
    managementGroups/XebiaRoot/providers/Microsoft.
    Authorization/policySetDefinitions/InheritTaggingFrom-
    ResourceGroup"
  management_group_id = azurerm_management_group.this.id
  # Both location and identity are needed for resource
  # changes when the policy contains "modify" or "deployIf-
  # NotExists" effects.
  # for this assignment location does not matter it will
  # only inherit the tags from the resource group
  location = var.policy_location
  identity {
    type = "UserAssigned"
    identity_ids = [
      azurerm_user_assigned_identity.policy_identity.id
    ]
  }
}
}

```

Exemptions

Sometimes, it is not possible to make the resources compliant. If needed, you can exempt certain policies for specific resources, resource groups, subscriptions, and event management groups. Exemptions are a way to exclude policies and ensure that only relevant policies are reported.

```

# Exemption Network Watchers resource location matches
resource group location on connectivity subscription
resource "azurerm_resource_group_policy_exemption"
"exemption_nww_location" {
  name                = "exemption_nww_location"
  resource_group_id   = azurerm_resource_group.network_
    watcher.id
  policy_assignment_id = "/providers/Microsoft.Management/
    managementGroups/123456-7890-0987-1234-123456789098/
    providers/Microsoft.Authorization/policyAssignments/
    LocationMatch"
  exemption_category   = "Mitigated"
}

```

Architectural Decisions Records (ADR)

This is a bit of a side step on the landing zones; it, however, is an important concept that comes in handy when building a landing zone or any other product. An Architectural Decision Record (ADR) is a document that captures an essential architectural decision made along with its context and consequences. It is a way to document the rationale behind decisions, ensuring that the reasoning is preserved for future reference. This can be particularly useful when revisiting decisions or when new team members must understand why certain choices were made.

ADR consists of:

- Title: A short, descriptive name for the decision.
- Context: Background information and the problem statement that led to the decision.
- Decision: The actual decision that was made.
- Consequences: The positive and negative outcomes of the decision.
- Status: The current status of the decision (e.g., proposed, accepted, deprecated).

Make sure to place the ADRs in a central place so everyone can read them and understand why certain decisions have been made. An ADR is not something that can never be changed; it is only a way of helping to understand the why behind it.

ADR 1: Use PostgreSQL for the Database

Context
Our application needs a robust, scalable, and open-source database solution.

Decision
We will use PostgreSQL as our primary database.

Consequences

- Positive:
 - PostgreSQL is highly reliable and has strong community support.
 - It supports advanced features like JSONB, which can be useful for our application.
- Negative:
 - Team members need to get familiar with PostgreSQL-specific features and configurations.

⁴ <https://github.com/Azure/Enterprise-Scale/>

Deployment

Workload Landing Zone Deployment

So, what happens if a workload team needs a subscription? Would it be smart to manually give them a subscription and let them figure out how to set it up properly? Probably not. If you want to help a workload team get started, you are better off helping them kick-start into Azure. In addition to helping the team, it will also help you make sure everything is compliant and connected when the team gets the subscription.

So, by default, we will deploy a virtual network for every new subscription connected to the Virtual Hub with the correct route tables on the two subnets we create. The virtual network also has the correct DNS Settings. Doing this ensures the connection is in place and already working. I can hear your thoughts, but what if I need more subnets? You are free to indicate the network size and add more subnets if necessary.

Besides the virtual network, we will provide a Key Vault, Storage Account, Log Analytics workspace, and default alerting. All the resources are compliant by default and deployed with IaC. The resources, subscriptions, and management groups are deployed using IaC. If needed, we can even deploy policy exemptions for resources. All these resources are what we call a "spoke canvas."

So now we have a management group, a subscription, and resources, but we also need one more thing: permissions. With our zero trust setup, we use the least privilege for everyone. The team will get a "reader" group and an "owner" group to achieve this. The reader group contains everyone from the team. By default, the owner group contains no members. If an incident happens, more permissions are sometimes needed. Privileged Identity Management (PIM) is here to help. PIM offers you the option to get elevated permissions for a short period. The "newest" setup is using groups. In our case, if you are a member of the reader group, you can 'pin' into the owner group to get your temporary permissions. PIM requires a reason and (highly recommended) approval to become active.

What is a workload? A workload is generally defined as an application or a value stream.

Layered deployment

As described earlier, we release a new version of a module or template when we modify the code inside it. These releases are then consumed by either Templates or Solutions. We use semantic versioning. This scheme uses three numbers separated by a dot — MAJOR, MINOR, and PATCH. It communicates the reason for the changes, for example, 2.1.3.

We do this to prevent big-bang module/template upgrades. Let take an example, we have a template called Windows Virtual Machine. Sadly we have to introduce a breaking change in the next release. Without versioning all solutions using this template would automatically consume the new version. This would results and a broken pipeline, cause we introduced a breaking change. However, with versioned templates and modules, we can control when to upgrade. If a breaking change occurs, we can modify the code before selecting the newest version.

However, this concept introduces some overhead. If 30 templates and 100 solutions rely on a single module, we need to update and merge version changes across 130 repositories. Luckily, we can borrow an effective practice from software development called automated dependency updates. This tool scans your repositories to check if a dependency (like a Terraform module) is up to date. If it's not, it automatically generates a pull request with the necessary version change. In theory, with automated tests in place, we could fully automate patch updates and potentially even minor version changes.

When you use Terraform or any other language to deploy your infrastructure, you would like to see fast and small deployments. When we introduced the Landing Zone, we also introduced the three parts (Connectivity, Management and Identity). Imagine you have a pipeline to deploy all the parts in one go. That may be fine for a single region, but what would happen if you had eight regions? Do you want to deploy all eight areas together? The answer to this question is probably "no."

The concept of Layered deployments helps us here. When using a layered deployment, you will split your deployment into smaller parts, which we call layers. In this case, we split the Landing zone into five parts: baseline, connectivity, management, identity, and diagnostics. The baseline will deploy some base resources we need to set up the other parts of the landing zone, like the DNS zones. We will deploy all the necessary resources for these landing zone parts in the connectivity, management, and identity layers. The diagnostic layer will deploy all the diagnostic settings on the resources.

This layer can be deployed separately to each region. So instead of having one big deployment, we now have eight times five equals forty small deployments. I can hear you think, so what's the difference? I still need to wait for the whole pipeline to complete. And if we leave it like this, then you are correct.

Luckily, because we created the layers, we can do something clever with those layers. We call it a Matrix deployment.

With a Matrix deployment, you will run only the layers that have been changed. In order to detect the change you need to create a script that finds the changes for the new deployment. When you use IaC, you will also need git as a version control tool. Git comes with many powerful and handy features, one of which enables you to get the changed files. So if a file in identity deployment changes, we can, based on that change, only run the identity layers, making our deployment much more minor.

Now, let's take a look at our eight environments. If something changes in West Europe, do you want to run all the other seven environments as well? If we combine our layered deployment with the environments we have and put both in a matrix deployment, we will end up running only West Europe or only North Europe because this is where we changed something. This way, we will save a huge amount of time when deploying the landing zones.

Innersource

Microsoft Azure has a lot of different services, so not all building blocks will be available. To ensure the teams are supported, the principle of Innersource is adopted. If no building block is available or existing building blocks need an update, teams are allowed to change or add Terraform templates and modules to the Platform Team repository. The platform team will verify if the changed or added templates apply to all the policies (four-eyes principle) and approve the merge request. After the approval of the changes, a new version of the template will be released. This puts the platform team in complete control of the building blocks without needing to change everything themselves.

Agents or Runners

When you use IaC, you will eventually need some tools to deploy the code to Azure. Weapons of choice are GitHub or Azure DevOps. Both use agents/runners to run the IaC and deploy your resources to Azure. The agents or runners are usually virtual machines hosted in Azure. And that brings us to an exciting part. You will need a subscription to host your agents on a Virtual Machine, but to deploy your code, you will need an Agent. This chicken-and-egg problem can be solved by initially creating the subscription by hand and the virtual machine from your local laptop using Terraform.

i NOTE

After each deployment the agents/runners are cleaned to make sure no code is "left behind" on the agent that contains sensitive information. This also aligns with the zero trust principles of the landing zone

There is one more thing you need to think about. The location where you deploy your runners. Do you want them to be outside or inside the landing zone? If you deploy them outside your landing zone, you can deploy the resources, but if you need a storage container inside a storage account or a secret in the key vault, it won't be easy. Remember, the firewall closes the landing zone, uses private connections, and isolates networks. On the other hand, if you deploy them inside the landing zone and they become part of the network, this problem is solved. But if you can create multiple landing zones, for example, a sandbox to play around, a nonproduction to test, and a production landing zone, you will also need agents inside each landing zone. If you only have an agent in the production landing zone, it will not be able to connect to the sandbox landing zone, and vice versa.

We ended up with agents inside and outside the landing zone just to ensure we could do everything we needed. Remember that you also need to set up your workflows in GitHub and Pipelines in Azure DevOps so that it knows what agent to use. That might be a different agent for Sandbox, production, and production.

Conclusion

Building a new Azure Landing Zone from scratch is a lot of work. It can be hard to figure out where to start and how to do the setup. The most crucial step is to determine what you need. The following steps are to build up the landing zone slowly. Do it in small steps because the only thing you get from a big bang is a big bang. These small steps can maybe deploy one resource at a time. It will be much easier to handle these changes and adjust anytime. And trust us, you will not get it right the first time; there will always be something that does not work as expected. You will also see that you are never actually 'done' with building a landing zone. There will always be a need for improvements.

This article should help you set up your landing zone and give you some insight into how we did it.

Enjoy your build!

Stop Creating Content With ChatGPT!

This year, I had the pleasure of speaking at NDC Oslo. I got to deliver a session on a topic I'm very passionate about: using different forms of generative AI to generate self-guided meditation sessions. You can read about it in XPRT Magazine #16^[1]. The day before the conference, I attended a community event in an attempt to unwind. At that event, I met one of the organisers of NDC Oslo. He asked what topic I was speaking on, and out of 100 sessions at that event, he knew precisely what session I was giving. And not for a reason I'm proud of, you see, I submitted a session abstract that I created with ChatGPT.

Author Matthijs van der Veer

I was happy enough with the result that I immediately submitted the abstract instead of reviewing it closely. Writers will tell you: "write drunk, edit sober". And reading the abstract back, I realised that ChatGPT must have been drinking at the time. The result was that I was roasted, in public, for my session abstract by the event organiser for fifteen minutes straight. And how did he know? Well, here's the first paragraph of the abstract:

In an era where technology and mindfulness intersect, the power of AI is reshaping how we approach app development. This session delves into the fascinating world of utilising artificial intelligence to expedite and streamline the development process of a mobile meditation app. We'll explore how Azure AI Speech, DALL-E, Azure OpenAI, and GitHub Copilot converge to eliminate the need for visual designers, voice actors, and sound designers, thereby revolutionising the traditional development workflow.

Can you see the telltale signs of (Chat)GPT? The buzzwords, the overpromising, the lack of focus? ChatGPT loves to "delve" into things. It will also start every abstract with "In a world / In an era". That the session was even accepted was a miracle. The abstract is too verbose, too buzzwordy, and oozed a lack of focus. It's not the kind of content I wanted to be known for.

I learned from that experience. Don't write your session abstract with ChatGPT. Don't take the easy route. Don't be lazy. Instead:

Build An Automated Abstract Generator With GitHub And Prompty

I'm convinced that we can create better content through Large Language Models (LLM). It levels the playing field and makes writing good content more accessible for people like me. People who like to talk and present and not write anything down. People who are not native speakers. People who are not writers. So I doubled down and built a system to help me generate better session abstracts. I used GitHub Actions to automate the process, and Prompty to generate the content. Here's how I did it:

I created a new issue template on my GitHub repo^[2] to get started. An issue template is a YAML file that will be used as a template when creating a new issue. This will force me to use a specific format whenever I want to generate a new session abstract. The template contains four fields that answer the following questions:

1. Who do you think this talk is for?
2. What do you think you'll learn from this talk?
3. What's something you'll be able to accomplish with the information gained from this talk?
4. What is the two-sentence summary of the talk?

¹ <https://pages.xebia.com/xprt-magazine-16-protecting-tomorrow-infuse-innovation>

² <https://github.com/MatthijsvdVeer/MondrianMuse/>

These questions came from a very talented speaker: Arthur Doler. He mentioned that he asks himself these questions when he creates his session abstracts. And I agree: If you can't answer these questions, you're not ready to submit your session abstract.

If you're working with LLMs and have yet to learn Prompty, give it a try^[3]! Prompty is a VS Code extension allows you to write prompts for LLM combined with the settings and examples needed for that prompt. This converges all your prompt-related settings in one file, which allows you to track changes over time in your git history. Next to that, Prompty comes with a rich dev and test experience. I won't cover all of Prompty's features in this article. It's a great tool for developing any application that leverages LLMs. It also seamlessly integrates with Prompt Flow, Langchain and Semantic Kernel, all the leading LLM orchestration tools. After installing the extension, simply right-click in VS Code's Explorer and choose "New Prompty". Below is the Prompty I created for generating a new session abstract:

```
---
name: CreateAbstract
description: A prompt that uses a set of questions and answers to create a new presentation abstract.
authors:
  - Matthijs van der Veer
model:
  api: chat
  configuration:
    type: azure_openai
    azure_endpoint: ${env:AZURE_OPENAI_ENDPOINT}
    azure_deployment: gpt-4o
  parameters:
    max_tokens: 3000
    temperature: 0.7
sample:
  answers: >
    <example answers go here>
---
system:
You are an expert in creating presentation abstracts. Our award-winning way of crafting an abstract is to ask a series of questions and use the answers to create a compelling abstract. I will give some examples of abstracts I like. Please match the wording, style and energy of the examples when crafting new ones.

## Examples:
Input:
### 1. Who do you think this talk is for?

Mostly developers, or others interested in automation

### 2. What do you think you'll learn from this talk?

How to use GitHub automation for content generation
The role of humans in reviewing GenAI output

How to chain different forms of GenAI to create unique content
```

```
### 3. What's something you'll be able to accomplish with the information gained from this talk?
```

```
Use GitHub actions to automatically create PRs
Use Azure AI services from a pipeline
```

```
### 4. What is the two-sentence summary of the talk?
```

```
Learn how to use GitHub actions to automate more than continuous integration or deployment. Leverage the GitHub's powerful platform the generate new and exciting content with Azure AI Services, while maintaining responsibility as a human.
```

Output:

```
# Automating Content Generation with GitHub Actions & Azure AI
```

```
In this session, you will learn how to harness the power of GitHub automation for content generation, leveraging GitHub Actions and Azure AI Services. Discover the role of human review and oversight in reviewing GenAI output and how to chain different forms of GenAI to create unique content. We'll explore a practical example of creating self-guided meditations using GPT-4, Azure AI Speech, and DALL-E 3.
```

```
This session is ideal for developers and anyone interested in automation. Expect many practical demos, including using GitHub Actions to automatically create PRs and integrating Azure AI services into your pipeline.
```

```
## What to avoid
```

- Never discriminate against any group of people
- Never use the words "delve", "equip", "empower", "navigate", "landscape", "enhance", "delve", "insight".
- Avoid terms like "in a world", or "in an era"

user:

```
{{answers}}
```

A Prompty file starts with some metadata and configuration. If you've worked with LLMs before, you will recognize some of the settings. In the file, I ask to use the **chat** API rather than the **completion** API and specify the deployment of my LLM. We also get to specify some parameters, like the temperature. This parameter is a value between 0 and 1, where 0 means the LLM will only generate the most likely text. The higher you set the temperature, the more unexpected the output will be. It allows the LLM to select less likely text, which resembles something that humans recognize as "creativity". I'm setting the temperature quite high, because this creativity can benefit my abstracts a lot. This is safe to do because I include examples of the output I expect in the prompt. Next to these settings, you can also add example input in the **sample** section. This allows us to execute the prompt locally and see the results. It's also a good example for others to see how the input should be structured.

³ <https://github.com/microsoft/prompty>



A few things should stand out in the the second half of the file. I start off by giving instructions on the role that I want the AI to take when creating the abstract. This gives the LLM more context on the job I'm trying to make it do. I also include instructions to match the style of the examples. This is an important technique in prompt engineering, known as a "few-shot prompt", where we ground the model's response with some examples. If I would leave out the examples, you will get responses based solely on what the LLM thinks a session abstract should be, with all the GPT-quirks of my original abstract. In the prompt I shared, I only included one example, but the more you add, the better the LLM will copy your intended style and wording. At the time of writing, my prompt uses two examples of abstracts I wrote without AI, in combination with the questions and answers I described above. This grounds the model enough to match my style of writing, as well as the overall length and tone of the abstract (I prefer short abstracts with a positive tone).

The prompt also includes things that I want the assistant to avoid. This is important, because without it we fall prey to the same mistakes I made with ChatGPT. Some of ChatGPT's behaviour comes from the system prompt that's invisible to us, but a big part of the output comes from using the same model that we're using here: GPT-4o. By telling the model what to avoid, we can prevent some of the same mistakes from happening. In prompt engineering, we usually favour positive examples over negative examples, but in practice you need to include guardrails and limitations.

If you wonder why I exclude that specific list of words, it's because they're overused in session abstracts starting around the time ChatGPT came out. Watch "Delving Into The Landscape"^[4] by Dylan Beattie on the subject.

GitHub Actions

Armed with my new prompt, I set out to automate the process. I created a new GitHub Workflow that runs every time a new issue is created with the **abstract** label. I won't focus too much on how to build such a workflow, please go ahead and steal mine^[5]. Most importantly, I need a way to run this Prompty file in a GitHub Action. Normally, I would default to a C# implementation. The code would need to be compiled first, and I will likely get stuck picking the best argument parser library and never get anything done. Luckily, Prompty steps in for the rescue again. I right-clicked on the Prompty file in VS Code and chose "Add Prompt Flow Code". This generates a small Python script that runs your Prompty file using Prompt Flow. You can do the same for Langchain (Python) and Semantic Kernel (C#), but Prompt Flow is definitely the least complicated if you're just running a simple prompt and don't want to compile anything. Don't know how to write Python? Can I suggest GitHub Copilot?

This makes the implementation in our workflow quite simple. After installing the Python dependencies and logging on to Azure, here's everything we need to get our abstract as a comment on the issue:

```
- name: 📝 Create an abstract
  env:
    AZURE_OPENAI_ENDPOINT: ${ secrets.AZURE_OPENAI_ENDPOINT }
  run: |
    python create-abstract_promptflow.py --answers
    "${ github.event.issue.body }" > abstract.txt
- run: gh issue comment $ISSUE --body "${cat abstract.txt}"
  env:
    GH_TOKEN: ${ secrets.GITHUB_TOKEN }
    ISSUE: ${ github.event.issue.html_url }
```

This GitHub workflow will run the Python script, save the output to a file and then use the GitHub CLI to post a comment on the issue with the generated abstract. The result is a generated abstract that's posted as a comment on the issue. This allows me to review the abstract, which brings us to the next problem: Automation Bias.

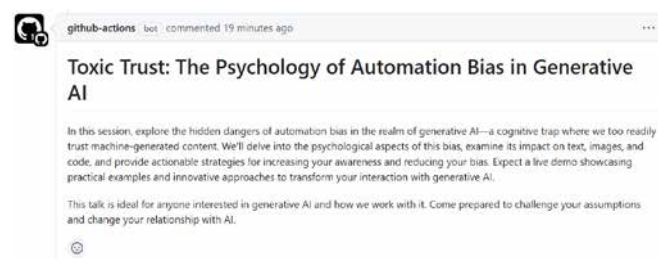


Figure 1: Screenshot of a generated abstract

Automation Bias: Review Everything

Like many people, I suffer from automation bias. This specific form of cognitive bias is the tendency to favour information generated by automated systems over information provided by humans. At the heart of my faulty abstract was not just content I generated with ChatGPT, but also the fact I didn't review it close enough. I saw words appear on screen that looked like an abstract, and I submitted it. And while more automation is likely not the answer, I doubled down and added even more automation. By now that should not surprise you, dear reader.

I decided to introduce two LLM-powered review tasks. One task reviews the content of the abstract. It checks if all the user-provided answers made it into the abstract and some general insights into the content. The second task is a style check. This prompt checks if the abstract is "too ChatGPT" for my taste. It detects most of the buzzwords and phrases I want to avoid. This way, I can review the abstracts generated by Prompty, and make sure they're up to my standards.

⁴ https://www.youtube.com/watch?v=Hqs24Gm0y_g

⁵ <https://github.com/MatthijsvdVeer/MondrianMuse/>

Both prompts follow the same structure as the prompt for generating abstracts. They start off with a style and instructions and are followed up with examples. A big difference is the **temperature** was set to 0.2, making sure the LLM doesn't become too creative in its answers but sticks to the task at hand. Remember: "write drunk, edit sober". I also instruct the model to make ample use of emojis. This way, the review will attract my attention when something is wrong. Here's an example:

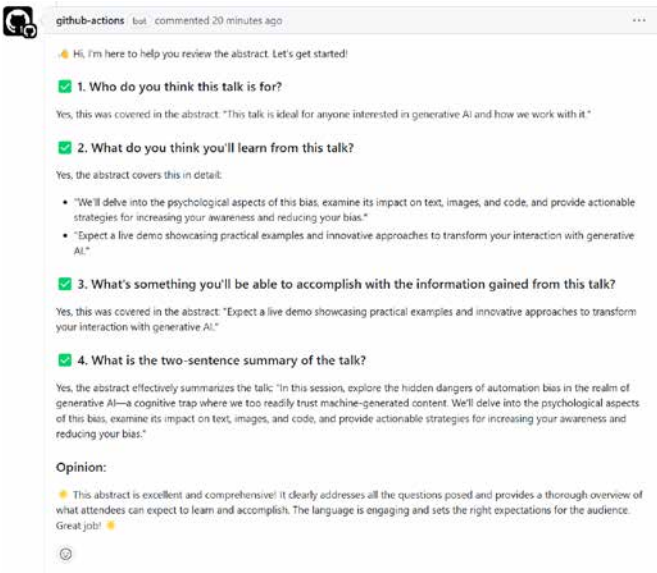


Figure 2: Screenshot of a generated review

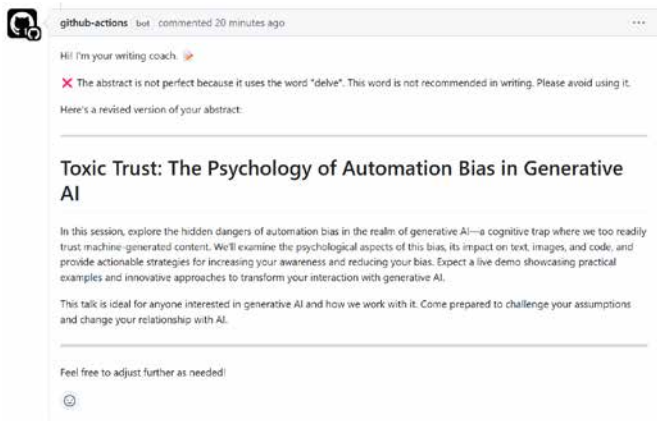


Figure 3: The writing coach enters the room

In the screenshots, you can see the reviews are quite thorough. The writing coach even correctly detected a mistake and then rewrote the abstract with only a tiny change. The reviews have been a great help. I still need to read the original text, but the reviews give me a good idea of what to expect. They also help me to spot mistakes I would otherwise miss. And even better: the first abstracts I've created this way are already being accepted at conferences.

Conclusion

Generalist tools like ChatGPT are great for a lot of things. But if you're going to create content in a professional setting, you're better off using specialist tools or building your own! I would love to say there's no replacement for a human review, and I believe that, in the case of Generative AI, that's still true today. But that was also true for spellcheckers, autocorrect, UI tests and many other tools we now rely on. I believe we can create better content with generative AI with the right prompts, the right guardrails, and the right review process.

I believe LLMs can even the playing field for people who aren't able to write with the fluency of a native speaker. I'm using it as a tool to help me create better content, not to replace my own creativity. Every abstract that's generated holds my style, my wording, and my ideas. The conclusion of this article: make better prompts, put in the work, and above all: review everything that generative AI generates. You're the one responsible for the content, not the AI.



Democratizing access to AI through GitHub Models

GitHub is the place where developers live, share, and try out new things. With the boom in AI in the last year, more and more people want to play around with AI and see how they can integrate it into their application and tools. There are a lot of providers of AI models available, some have free tiers to get started with, and some have only paid access to their model library.

Author Rob Bos

An AI model is a program that has been trained on a set of data to recognize certain patterns or make certain decisions without further human intervention. It can be used to generate things like text, images, or even code. A common example of this, that most people have heard of, is ChatGPT, where you can ask it a question like "Create a poem on the beauty of a forest during sunset", and the model will produce a nice poem.

GitHub has created a free way to access the power of AI through their GitHub Models functionality. GitHub Models is a way to get access to AI capabilities through GitHub's user interface, that enables you experiment with AI to see what you can do with it. There is a playground available that lets you ask the AI model questions and get the answers back in your browser. GitHub Models contains a collection of pre-trained models that you can use in your application. These models are trained on a variety of datasets and are ready to be used and come from different providers: from Meta-Llama to OpenAI, Cohere, Mistral, to Phi. The main benefit is that you can use your GitHub Access Token to authenticate against the models. That gives you plenty of room to test and validate your ideas without any additional cost.

Let's dive into models!

NOTE

At the time of writing GitHub Models is in private beta, so some changes are expected to happen in the future.

Marketplace

The GitHub Marketplace is the place to find loads of tools offered by the community. Most of it is free (GitHub Actions) and some have a paid subscription model in them as well (some GitHub Apps). GitHub Models is a new addition to the Marketplace and is free to use. You can start by going to the marketplace and select 'Models'^[1]. The GitHub Models on the marketplace are a subset of what is available in the Azure Open AI models.

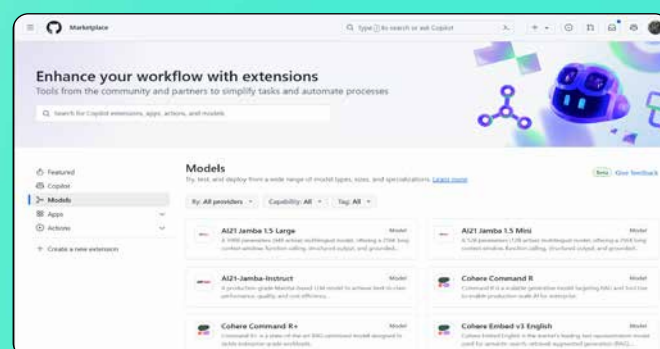


Figure 1: Screenshot of the GitHub Models on the marketplace

¹ <https://github.com/marketplace/models>

By using the filters you can search for models and learn more about them. Clicking on a card will show you the details for those models. This includes links to learn more about the model or the way it was trained.

Playground

By going to the playground (upper right hand corner) you can now start testing around with the model. It already shows some examples out of the box to get you started with a chat conversation with the model you choose. Note that in the top left corner you can easily switch between different models!

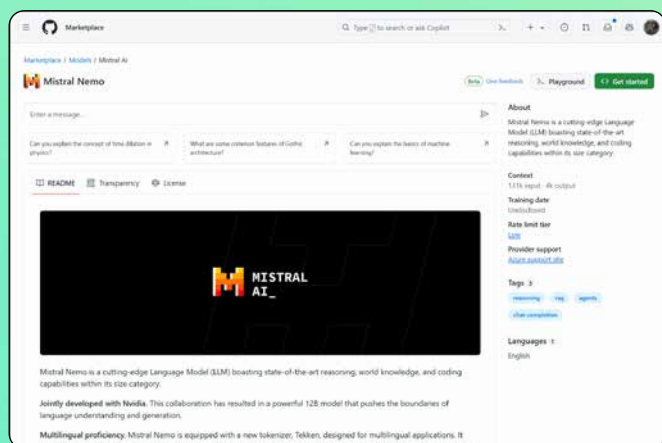


Figure 2: Screenshot of the Mistral Nemo model

Even better, all the code that you are working with is available in the playground as well. On the code tab, you can find the code that is needed to make the test in the playground work. This way you can easily copy and paste the code into your own application and start using the model in your own application. The playground shows you how to build your code against the model in several programming languages: Python, JavaScript, C#, and REST calls using cURL.

Azure OpenAI

GitHub Models has been created as an easy way to get started using AI. You can choose to use the SDK from the model vendor, or to use the Azure AI Inference SDK. The recommendation is to use the Azure SDK, as that abstracts away the complexity of the different model implementations. Instead of working with the vendor flavor, the Azure SDK lets you switch the model by just changing the model identifier.

In the example below you can see how to use the Azure SDK to interact with the model. The code is in Python and uses the Azure SDK to interact with the model. After setting up the client with authentication, we create a completion request with a system message (grounding the conversation in a certain direction). After that, we create a user message that we want to send to the model. This is called the "prompt". The last line shows the response from the model, which is the completion of the user message.

```
client = ChatCompletionsClient(
    endpoint="https://models.inference.ai.azure.com",
    credential=AzureKeyCredential(os.environ["GITHUB_TOKEN"]),
)

response = client.complete(
    messages=[
        SystemMessage(content=""),
        UserMessage(content="Can you explain the basics of machine learning?"),
    ],
    model="Mistral-Nemo", # change this line to use a different model
    temperature=0.7,
    max_tokens=4096,
    top_p=1
)

print(response.choices[0].message.content)
```

The way all this works is that GitHub is hosting the models on Azure, and is letting you authenticate against their implementation using your GitHub Token. You can use this authentication from anywhere, so not only through the playground, but also in your own application!

Even inside of a GitHub Codespace (a virtual workspace hosted by GitHub), the GITHUB_TOKEN environment variable in that workspace is automatically set to the token of the user using the Codespace. This way you can easily use the models in your Codespace without any additional setup.

Very awesome in my opinion! This means that ANYONE with a GitHub account can start using AI models in their solutions. Whether you want to build this into your mobile application or your web backend, you can try it out with GitHub Models! No need to sign up for a new service, no need to pay for a subscription, just use the models in your application and start testing.

Getting started

The getting started button will take you to a screen with more examples. The examples guide you on how to implement the setup you have been testing in the Playground and add them into your own application or repository.

When you use the button "Run Codespace", you will actually create a new Codespace on the **github/codespaces-models** repository. A Codespace is a full-fledged hosted coding environment, that saves you downloading the repo as well as installing all the dependencies. Every GitHub user has access to 60 hours of free Codespace usage per month (with a dual core config). You can use this to test out the models in your own environment, without running into any compute cost.

NOTE

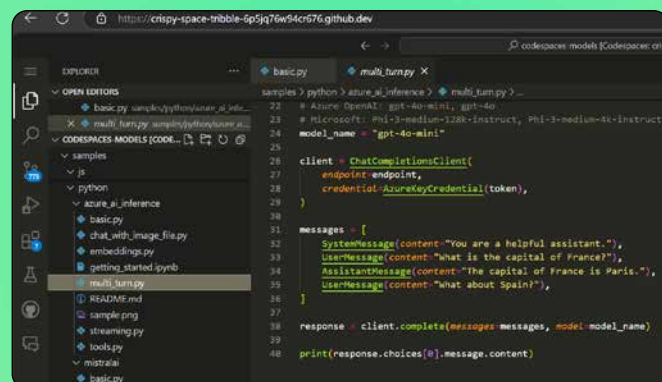
The Codespace is a completely separate experience from the Playground. The Codespace is configured with all sorts of sample projects for you get started with.

The Codespace is equipped with a lot of examples and documentation to get you started with using the models. You can easily run the examples and see how they work, right from the terminal in the Codespace. Since your GITHUB_TOKEN is automatically set in the Codespace, you can go straight to the terminal and run the examples.

Open up a folder in the code language that you prefer, and learn all about the different AI interaction options that are available to you:

- Basic chat (one prompt and response turn)
- Chat with an image file (ask questions on the image)
- Embeddings (retrieve a part of a text, like the city name from a sentence)
- Multi-turn chat (having chat conversations with history)

There are also several examples available in Jupyter notebook files (.ipynb) that you can run in the Codespace. A Jupyter Notebook is a common way in the machine learning community to document steps in a process and have a user interface to run those steps. The VS Code extensions to run the notebook are already installed in the



```

22 samples > python > azure_ai_inference > multi_turn.py >
23 # Azure OpenAI: gpt-4o-mini, gpt-4o
24 # Microsoft: Phi-3-medium-128k-instruct, Phi-3-medium-8k-instruct,
25 # model_name = "gpt-4o-mini"
26
27 client = ChatCompletionClient(
28     endpoint=endpoint,
29     credential=AzureKeyCredential(token),
30 )
31
32 messages = [
33     SystemMessage(content="You are a helpful assistant."),
34     UserMessage(content="What is the capital of France?"),
35     AssistantMessage(content="The capital of France is Paris."),
36     UserMessage(content="What about Spain?"),
37 ]
38
39 response = client.complete(messages=messages, model=model_name)
40 print(response.choices[0].message.content)
  
```

Figure 3: Example of getting started with multiturn

Codespace, so you can just open the notebook and run the cells to see the output. Click on the 'play' icon the execute the step. In the screenshot you can see the UI for executing the steps, with the description of what happens in the step, the actual code for the step, and the highlighted result of the step.

The benefit of the notebook is that these have been filled with a lot more guidance on what is happening, so this is a great way to learn more about the code that is needed for application or AI-flow.

Change the code in the notebook and see how the output changes. All inside a of a free environment.



Rate limits

Since this all is freely available there have to be some limits. The setup is not intended for production use, but for development and testing. The rate limits are very sufficient for you to start testing and give you plenty of room to work during the day by yourself.

As you can see in the screenshot, the amount of calls per minute and hour are dependent if you already have a GitHub Copilot license or not. The limit for the amount of tokens in and out of the model is limited to a relative straightforward amount as well: 12.000 tokens (in+out) is plenty to start experimenting and validating if your idea works.

Rate limit tier	Rate limits	Free and Copilot Individual	Copilot Business	Copilot Enterprise
Low	Requests per minute	15	15	20
	Requests per day	150	300	450
	Tokens per request	8000 in, 4000 out	8000 in, 4000 out	8000 in, 8000 out
	Concurrent requests	5	5	8
High	Requests per minute	10	10	15
	Requests per day	50	100	150
	Tokens per request	8000 in, 4000 out	8000 in, 4000 out	16000 in, 8000 out
	Concurrent requests	2	2	4

Figure 4: Rate limit overview

Going to production

If you have played around enough with the free options, it is time to start testing with a broader audience and validate your solution with other people (e.g. beta testers). If you have chosen the Azure AI Inference SDK, you can easily switch to a paid Azure OpenAI instance in your own Azure subscription. If you have chosen the vendor SDK, you can switch to a paid subscription with the vendor.

Deploying to Azure

To deploy your AI integration to Azure you need to first deploy an Azure OpenAI instance. You can deploy that the same way you would deploy any other Azure service: through the Azure portal, the Azure CLI, or through an ARM/bicep template.

Here's an example to get you started:

```
# example deployment of an Azure OpenAI instance using
the Azure CLI
az cognitiveservices account create \
  --name MyOpenAIResource \
  --resource-group OAIResourceGroup \
  --location swedencentral \
  --kind OpenAI \
  --sku s0 \
  --subscription <subscriptionID>
```



```
# deploy a model to your Azure OpenAI instance
az cognitiveservices account deployment create \
  --name MyOpenAIResource \
  --resource-group OAIResourceGroup \
  --deployment-name gpt-4o \
  --model-name gpt-4o \
  --model-version "2024-05-13" \
  --model-format OpenAI \
  --sku-capacity "1" \
  --sku-name "Standard"
```

When you have deployed the Azure OpenAI instance, get a key from the 'keys and endpoint' section of the Azure OpenAI instance and use that in your application. The GITHUB_TOKEN is no longer needed then. You only need to change the endpoint and the API Token in your application to start using the Azure OpenAI instance.

Here's the example of the new code with only two lines changed:

```
client = ChatCompletionsClient(
    endpoint="https://xms-openai.openai.azure.com/
    openai/deployments/gpt-4o", // this line was
    changed, do note the deployments/deploymentname,
    without /models/completions!
    credential=AzureKeyCredential(os.environ["AOAI_
    TOKEN"]), // this line was changed
)

response = client.complete(
    messages=[
        SystemMessage(content=""),
        UserMessage(content="Can you explain the basics
        of machine learning?"),
    ],
    model="Mistral-Nemo", # change this line to use a
    different model
    temperature=0.7,
    max_tokens=4096,
    top_p=1
)

print(response.choices[0].message.content)
```

Conclusion

GitHub Models is a great way to get started with AI models. It is free to use and you can start using the models in your application right away. The Azure OpenAI SDK is recommended to use, as that abstracts away the complexity of the different model implementations.

Be aware that the authentication happens through your GitHub Token, which enables you to run your tests from anywhere. You are not tied to the Playground, the Codespace, or running this in GitHub. I'm curious to see what you will build with GitHub Models!

Was shift left the right move?

Shifting left, by adopting DevOps practices, improves the performance of development teams. Granting teams the responsibility for all aspects of their software products allows them to be more efficient.

Authors Sander Aernouts and Chris van Sluijsveld

A study from Leiden University^[1] shows that the adoption of DevOps practices leads to improvement in both software quality and delivery speed. This is not surprising; The goal of shifting left, and DevOps, is to shorten the feedback loop and reduce lead time. In 2014 McKinsey^[2] already found that projects that adopt short cycle times tend to perform better, and deliver higher quality outcomes. To effectively shorten cycles, and improve the performance of development teams, autonomy is essential.

The downside of shifting left

The autonomy that comes with shifting left has a price. It has increased the load on development teams. Teams build and maintain all aspects of their software products. They are responsible for security, compliance, costs, sustainability, regular operations, and any other

aspect that comes with running applications. All these responsibilities take a lot of time and attention away from building software and creating business value, which is the purpose of the teams.

This high cognitive load negatively impacts productivity. It is hard to keep all aspects of the software product in mind all the time.

Cognitive load theory suggests that your brain has a limited capacity to process and store information at any given time. If the cognitive load is too high, your brain will not be able to process and store all the information. Puppet's state of DevOps report 2021^[3] even found that unbounded cognitive load has a negative impact on all performance indicators.

Silos

Development	Operations		Security	Finance	Compliance
Busines logic	ALM	Observability	Security	Cost	Compliance

Shift left

Development team					
Busines logic	ALM	Observability	Security	Cost	Compliance

¹ <https://arxiv.org/pdf/2211.09390>

² <https://www.mckinsey.com/~media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Achieving%20success%20in%20large%20complex%20software%20projects/Achieving%20success%20in%20large%20complex%20software%20projects.ashx>

³ <https://www.dau.edu/sites/default/files/Migrated/CopDocuments/Puppet-State-of-DevOps-Report-2021.pdf>



To meet all these demands each team has to solve problems. They need setup their developer environments, manage dependencies, deal with build failures, etc. All these tasks are necessary but do not directly contribute to the business's core objectives or innovation. This is what we call developer toil.

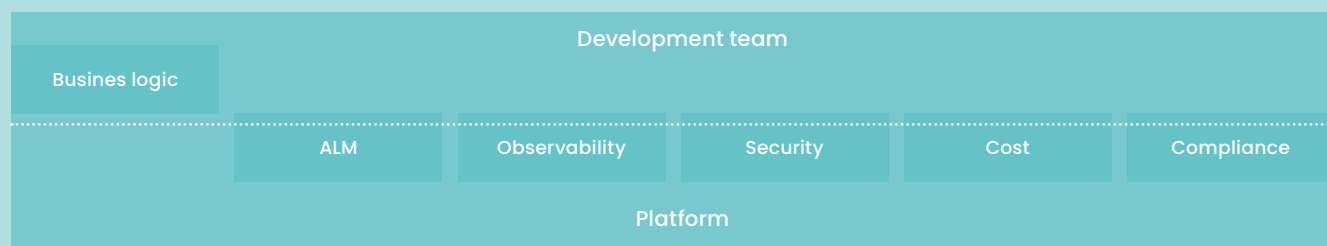
Like cognitive load, developer toil also has a big impact on productivity, innovation, and motivation. While some level of toil is part of the job, excessive toil can be detrimental. It consumes precious time, drains motivation, and can lead to burnout^[4].

So, whilst shifting left improves the performance of development teams by reducing cycle time, it also increases the cognitive load and developer toil for these teams. This hinders, or even partially negates, the benefits of shifting left.

Shift down

While keeping the benefits of shifting left, the goal is to find ways to reduce the cognitive load and developer toil on development teams. Ideally teams to spend their time on innovation and creating value, while keeping the autonomy that comes from shifting left. But instead of only shifting left, the next step would be to also shift down. Shifting down is

Shift down



⁴ <https://www.forbes.com/consent/ketch/?toURL=https://www.forbes.com/councils/forbestechcouncil/2024/03/20/developer-toil-is-a-problem-heres-why-companies-need-to-address-it/>

a term introduced by Google's Richard Seroter^[5]. It means splitting responsibilities horizontally by offering platforms for development teams which they can build upon.

At Google, they offer managed experiences for coding, testing, building, releases, roll-outs, hosting, alerting and more^[5]. These platforms are maintained by dedicated teams of engineers, so that development teams can focus again on building software and creating business value.

Platform engineering

Why not have special teams dedicated to building platforms that provide the infrastructure, tools, and processes that support software development across your organization.

Gartner expects around 80% of organizations plan to have a team dedicated to Platform Engineering by 2026^[6].

Platforms abstract complexity, integrate best practices, and provide reusable components, ultimately streamlining the development process. This way we can unburden the development teams from a lot of operational aspects and repetitive tasks.

To reduce developer toil and cognitive load, platform engineering teams have to tackle a few key challenges:

1. Automate Everything

Identify tasks that are repetitive and time-consuming and automate them. Continuous integration/continuous deployment (CI/CD) pipelines, infrastructure-as-code (IaC), and automated testing frameworks are great starting points.

2. Offer a golden path

Create well-defined combinations of tools and building blocks that meet all the organizational requirements and standards. These golden paths should be easy to use and provide a clear path to production without requiring deep knowledge of the underlying platforms. They should cater to the majority of the teams. Teams that require more flexibility or have specific requirements can still choose their own path.

3. Self-Service

Provide self-service experiences that allow developers to independently handle routine tasks, such as onboarding new team members, modifying shared infrastructure, etc. This reduces their dependency on other teams and speeds up the development process.

⁵ <https://cloud.google.com/blog/products/application-development/richard-seroter-on-shifting-down-vs-shifting-left>

⁶ <https://www.gartner.com/en/infrastructure-and-it-operations-leaders/topics/platform-engineering>



Platform as a product

Platforms should be treated as products in their own right. The platforms are built by specialized teams that are end-to-end responsible for building and maintaining the platforms, just like the other development teams. DORA's state of DevOps 2023 suggests that treating development teams as the users of their product is a more successful approach than build it and "they" will come^[7].

The platform team is responsible to ensure the platform is secure, compliant, sustainable, etc by default. They make sure the platform is easy to use and easy to understand while being cost-effective. In other words, make sure the platform is a solid foundation for other teams to build on. This way the platform team creates business value by reducing the time other teams have to spend on all the details that come with running software in the cloud.

Of course there can also be pitfalls with treating platforms as products. The platform team can become too focused on the platform itself and lose sight of the needs of the development teams. They can become too focused on the technical aspects and lose sight of the business value. The platform team should always keep the development teams in mind and make sure they are building a platform that is valuable for the development teams. It can also be resource intensive. Developing and managing a platform often requires substantial time, financial investment, and human resources, which can strain budgets and personnel.

Conclusion

Shifting left is the right move! Teams get empowered to build and run their software products. Through engineering platforms, you can provide a solid foundation to unburden the teams. This way teams can spend most of their time on what matters most: innovation and creating business value.

Breaking down the silos and empowering teams by shifting left, giving teams end-to-end responsibility, has also burdened these teams with a lot of extra work. Development teams are responsible for building and running their application. They are responsible for security, costs, compliance, sustainability, regular operations, and any other aspect that comes with running applications. All these responsibilities take a lot of time away from innovation and creating business value, which is the purpose of the teams.

Shifting down, by providing platforms, allows you to keep empowering your teams with end-to-end responsibility. Dedicated platform teams build and maintain these platforms. The platforms are products on their own. They create business value by unburdening other teams. This way you can keep the benefits of shifting left, while reducing the cognitive load and developer toil on development teams by shifting down.

Not sure how to start with platform engineering? Xebia has an accelerator to give you a head start with platform engineering on Azure. For more information on a technology that can help with platform engineering, check out Cloud-Native Application Development with Radius by Loek Duys in this same magazine.

References

- [1] Offerman, Blinde, Stettina, and Visser. "A Study of Adoption and Effects of DevOps Practices". arXiv, 17 Nov. 2022, pp 9. <http://arxiv.org/pdf/2211.09390>
- [2] McKinsey & Company. "Achieving success in large, complex software projects". McKinsey Digital, July 2014. <https://www.mckinsey.com/~media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Achieving%20success%20in%20large%20complex%20software%20projects/Achieving%20success%20in%20large%20complex%20software%20projects.ashx>
- [3] Puppet. "State of DevOps report 2021", Puppet, 2021, pp 16. <https://www.dau.edu/sites/default/files/Migrated/CopDocuments/Puppet-State-of-DevOps-Report-2021.pdf>
- [4] Forbes. "Developer Toil Is A Problem—Here's Why Companies Need To Address It", Debo Ray, 20 March 2024. <https://www.forbes.com/councils/forbestechcouncil/2024/03/20/developer-toil-is-a-problem-heres-why-companies-need-to-address-it/>
- [5] "The Modernization Imperative: Shifting left is for suckers. Shift down instead", Richard Seroter, 9 June 2023. <https://cloud.google.com/blog/products/application-development/richard-seroter-on-shifting-down-vs-shifting-left>
- [6] "Platform Engineering That Empowers Users and Reduces Risk", Gartner. <https://www.gartner.com/en/infrastructure-and-it-operations-leaders/topics/platform-engineering>
- [7] "State of DevOps Report", DORA, 2023, pp 19. <https://dora.dev/research/2023/dora-report/>

⁷ <https://dora.dev/research/2023/dora-report/>



The future of Cloud-native software development with Radius

The rise of platform engineering

Over the years, the process of software development has changed a lot. The way applications are built, deployed, and managed today is completely different from ten years ago. Initially, our industry relied on monolithic architectures, where the entire application was a single, simple, cohesive unit. This approach made the development process straightforward initially, but as applications grew in complexity, maintaining and scaling them became increasingly challenging. Every change required a complete redeployment, leading to long development cycles and heightened risks of introducing errors. On top of that, a single bug in the software could take down an entire system.

Author Loek Duys

Ever increasing complexity

To overcome these limitations, we transitioned to Service-Oriented Architecture (SOA). SOA decomposed applications into smaller, independent services that communicated over a network. This modular approach improved maintainability and scalability of applications, as each service could be developed, deployed, and scaled independently. However, SOA introduced its own set of complexities, such as the need for robust inter-service communication and service management. On top of that, services depending on multiple layers of other downstream services resulted in cascading errors, so a single issue could still result in large scale unavailability.

DevOps

The introduction of DevOps marked a cultural and operational shift in software development. DevOps emphasized the collaboration between development and operations teams, breaking down silos and fostering a culture of continuous integration and continuous delivery (CI/CD) and an Agile way of working. This approach enabled faster, more reliable and efficient software delivery

by automating infrastructure management and the deployment processes. The focus on collaboration and automation greatly improved the efficiency of software development. Adopting a DevOps culture did increase the overall cognitive load for team members, as they need to learn about CI/CD and automation.

Microservices

Building on the principles of SOA, Microservices architecture further decomposed applications into self-contained autonomous business capabilities. Each Microservice focused on a specific business function and could be independently developed, deployed, and scaled. This granularity improved agility and scalability but also introduced challenges in managing service dependencies, communication, and data consistency, further increasing the cognitive load for team members.

Containers and orchestration

The use of containers greatly simplified application deployment, by packaging an application together with its direct dependencies like libraries, frameworks and content

files. Running containers reliably at scale lead to the introduction of container orchestrators like Kubernetes. Kubernetes is able to run containerized workloads on a cluster of virtual machines, and provides many additional features. DevOps teams did need to learn how to containerize their apps and how to deploy them to an orchestrator, again increasing complexity.

Cloud

Around the same time, the Cloud became more and more popular as an environment to run software. We started building Cloud-native software. Using IaaS and PaaS services instead of custom built self-hosted tools greatly accelerated teams. The downside was that they first needed to understand which Cloud service to use when. Not an easy task considering that Azure has more than 200 services and products at the time of writing.

Light at the end of the tunnel

Today, we have arrived at platform engineering, a field that takes the best practices and tools from previous methodologies to optimize the development, deployment, and management of applications. This is a significant step in lowering cognitive load on product teams. Platform engineering provides a standardized environment that integrates tools and processes to enhance collaboration and efficiency. It abstracts many of the complexities of underlying infrastructure, enabling developers to focus on delivering features and value rather than managing operational details. [If you want to read more about Platform Engineering, have a look at the article 'Was shift left the right move?' by Sander and Chris, also in this magazine!]

This is where Radius comes into the picture. Radius is designed to simplify the development and deployment of Cloud-native applications. The core feature of Radius, is the application graph, which represents the relationships and dependencies within applications. It can be visualized in the Radius Portal which is installed with the product, or the Radius CLI. This enables collaboration between people with a development role and people with an operations role.

What is Radius and how does it help developers?

Radius was originally developed by Microsoft's incubation team. Nowadays, it is an open-source project and part of CNCF. Radius is designed to address the challenges of modern Cloud-native software development. As Cloud-native architecture becomes the standard for many IT companies, managing the complexity of applications across multiple environments can be complicated. Radius provides a platform that simplifies the entire lifecycle of Cloud-native applications. It bridges the gap between developers and operators, enabling collaboration.

Applications

One of the key features of Radius is the Application graph. Graphs visually represent the relationships and dependencies between different components of an application, like compute, data storage, messaging and networking. This visual approach simplifies the understanding and management of complex applications. By mapping out how various services interact, developers can quickly understand how they work. The application graph also makes it easy to see how to operate the application, which dependencies should be implemented as PaaS services and which as containers.

In Radius applications are defined using the Bicep language. Up until now, Bicep used to be domain-specific language for Azure resource deployments. It has now been extended to include Radius resources. In the fragment below, you can see a simple Radius application. It defines an Environment which specifies a lifecycle stage for the application. It also contains an Application definition, and as part of the application and environment it runs one containerized web server. [A sample application created by the Radius team.]

```
//import Radius resource types
extension radius

//define radius environment
resource env 'Applications.Core/environments@2023-10-01-preview' = {
  name: 'test'
  properties: {
    compute: {
      kind: 'kubernetes'
      namespace: 'test'
    }
  }
}

//define radius application
resource app 'Applications.Core/applications@2023-10-01-preview' = {
  name: 'demo01'
  properties: {
    environment: env.id
  }
}

//define container that runs the application
resource container01 'Applications.Core/containers@2023-10-01-preview' = {
  name: 'container01'
  properties: {
    application: app.id
    environment: env.id
    container: {
      image: 'ghcr.io/radius-project/samples/demo:latest'
      imagePullPolicy: 'IfNotPresent'
      ports: {
        web: {
          containerPort: 3000
        }
      }
    }
  }
}
```

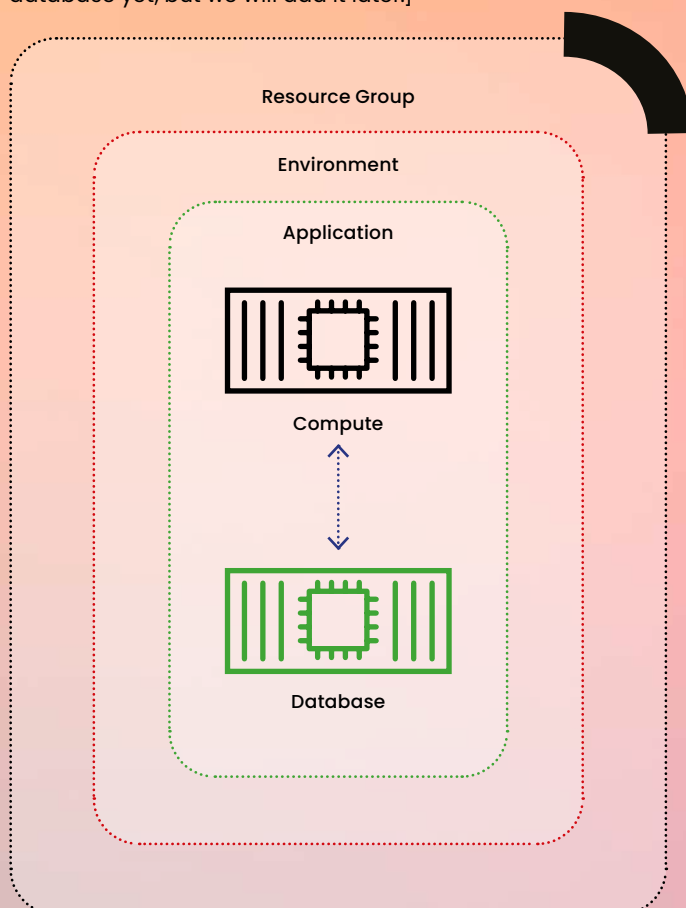
Environments

In Radius, environments are used to manage the lifecycle of cloud-native applications. Environments in Radius represent distinct stages such as development, testing, staging, and production, each tailored to a specific phase of the application lifecycle.

Resource groups

Radius Resource Groups are logical containers that help manage and organize the resources required for deploying and running applications. These groups contain various resources such as Environments, containers, databases and networking components, all necessary to run a specific workload. Radius utilizes Resource Groups to streamline the deployment process, making it easier to manage resources collectively rather than individually. In the future, Radius Resource Groups will also act as a security boundary by applying Role Based Access control policies at this level.

The image below visually represents the relationship between Application components, Environment and Resource group. [The first demo app does not have a database yet, but we will add it later.]



Running the demo

If you want to run this and other examples yourself, the quickest way to get up and running is by creating a GitHub Codespace^[1]. This article uses v0.37. Run this command to fix an issue in the Codespace:

```
@loekd → /workspaces/samples (v0.37) $ sh ./
.devcontainer/on-create.sh
```

This should install the proper Radius CLI version.

The bicep file of the app above^[2]. Copy the files to your Codespace before deploying them. For example, by using curl:

```
@loekd → /workspaces/samples (v0.37) $ curl -O
https://raw.githubusercontent.com/loekd/radius-demos/
main/01-Bicep/app_v1.bicep
```

Also, make sure to enable the extensions preview for Bicep in 'bicepconfig.json'. Example:

```
{
  "experimentalFeaturesEnabled": {
    "extensibility": true,
    "extensionRegistry": true,
    "dynamicTypeLoading": true
  },
  "extensions": {
    "radius": "br:biceptypes.azurecr.io/radius:0.37"
  }
}
```

Check the image version for the Radius extension.

Prepare Radius for the application. Choose k3d-k3s-default as the target Kubernetes cluster (K3d running locally). Create an environment named 'test' and a namespace named 'test'. Don't configure any cloud features, and don't set up an application (as this is already done using the Bicep code above).

```
@loekd → /workspaces/samples (v0.37) $ rad init --full
```

Initializing Radius. This may take a minute or two...

- ✓ Install Radius v0.37.0
 - Kubernetes cluster: k3d-k3s-default
 - Kubernetes namespace: radius-system
- ✓ Create new environment test
 - Kubernetes namespace: test
- ✓ Update local configuration

Initialization complete! Have a RAD time 😊

You can then run your Radius application using the rad run command:

```
@loekd → /workspaces/samples (v0.37) $ rad run
./app_v1.bicep --application demo01 --group test
Building ./app_v1.bicep...
```

Deploying template './app_v1.bicep' for application 'demo01' and environment 'test' from workspace 'default'...

¹ <https://github.com/codespaces/new/radius-project/samples>

² https://github.com/loekd/radius-demos/blob/main/01-Bicep/app_v1.bicep



Useful links:

[1] Radius documentation:
<https://docs.radapp.io/>

[2] Radius GitHub:
<https://github.com/radius-project>

[3] Radius sample applications
from this article: <https://github.com/loekd/radius-demos>

Deployment In Progress...

```
Completed      demo01      Applications.Core/
applications
Completed      test       Applications.Core/
environments
...            container01 Applications.Core/
containers
```

Deployment Complete

Resources:

```
demo01      Applications.Core/applications
container01 Applications.Core/containers
test        Applications.Core/environments
```

Starting log stream...

```
+ container01-b5b9bf6bc-657p4 > container01
container01-b5b9bf6bc-657p4 container01 [port-forward]
connected from localhost:3000 -> ::3000
container01-b5b9bf6bc-657p4 container01 No APPLICATION-
INSIGHTS_CONNECTION_STRING found, skipping Azure Monitor
setup
container01-b5b9bf6bc-657p4 container01 Using in-memory
store: no connection string found
container01-b5b9bf6bc-657p4 container01 Server is
running at http://localhost:3000
dashboard-5d64c96ff-9jwf7 dashboard [port-forward]
connected from localhost:7007 -> ::7007
```

The CLI is collecting output generated by the container, and displaying them in your terminal. Also, the CLI has created the necessary port forwards, so you can access the application running on the K3d cluster using `localhost`. Press Control and click on the URL `http://localhost:3000` in the output to see the web page running. It should show the text 'Welcome to the Radius demo'.

In your Codespace, hit Control + C to terminate the log and port forwarding

Next, remove the application by running the following script:

```
rad app delete demo01 --group test -y
```

Repeat this process for the next demo's.

Recipes

Radius also uses Recipes. Recipes enable a separation of concerns between IT operators and developers by automating infrastructure deployments for application dependencies. Developers can select **which types** of resources they need in their applications, such as Mongo Databases, Redis Caches, or Dapr State Stores, while IT operators define **how** these resources should be deployed and configured within their environment, whether it be as containers, Azure resources, or AWS resources. Before developers can use a Recipe, they must be versioned and published to an OCI-compliant registry like Azure Container Registry.

When a developer deploys an application and its resources, Recipes automatically deploy the necessary backing infrastructure and bind it to the developer's resources. Recipes supports multiple Infrastructure as Code (IaC) languages, including Bicep and Terraform, and can be referenced in an Environment. By using Recipes, a single application definition can be deployed to a non-production Environment with a containerized state store, and to production with a Cloud based store like Cosmos Db without changes, but simply by referencing different Recipes.

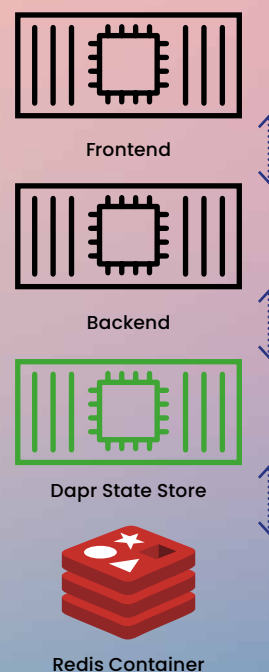
Connections

Recipes must follow some rules so Radius can connect dependencies created through Recipes to Application containers. For example, if you deploy a database using a Recipe, the container that uses that database will need to know how to connect to it. Radius solves this by requiring a Recipe to return details about the deployed resource, like connection strings, and/or credentials. These details are known as Connections.

In the script below, you can see a Dapr Application that uses a Recipe to define its Dapr State Store. The Recipe is referenced from the Environment definition. Also, it uses the concept of Bicep modules to reference resources from other files that will define the frontend and backend elements for the Application.

If you want to deploy this example, use the file `app.bicep`.

The image below visually represents the relationship between Frontend, Backend, Dapr State store and Redis Database. In this situation, the connection is configured for the Backend, so it knows which Dapr State store component to use.



app.bicep:

```

extension radius

//define explicit radius environment
resource env 'Applications.Core/environments@2023-10-01-preview' = {
  name: 'test'
  properties: {
    compute: {
    }
  }
  //register recipe using Bicep
  recipes: {
    'Applications.Dapr/stateStores': {
      default: {
        templateKind: 'bicep'
        templatePath: 'acrradius.azurecr.io/recipes/statestore:0.1.0'
      }
    }
  }
}

resource app 'Applications.Core/applications@2023-10-01-preview' = {
  name: 'demo02'
  properties: {
    environment: env.id
  }
}

module frontend 'frontend.bicep' = {
  name: 'frontend'
  params: {
    environment: env.id
    application: app.id
  }
  dependsOn: [
    backend
  ]
}

module backend 'backend.bicep' = {
  name: 'backend'
  params: {
    environment: env.id
    application: app.id
  }
}

```

In the fragment below, you can see the contents of the **backend.bicep** file. Notice that the Application has a Connection that references the State Store. The application does not need to know the concrete implementation of the Store, just how to connect. The State Store **stateStore02** is deployed by declaring the resource, and referencing the Recipe name **default**. Radius will use the Environment definition to find the template and ensure it is created.

backend.bicep:

```

import radius as radius

@description('Specifies the environment for resources.')
param environment string

@description('The ID of your Radius Application.')
param application string

// The backend container that is connected to the
Dapr state store
resource backend02 'Applications.Core/containers@2023-10-01-preview' = {
  name: 'backend02'
  properties: {
    application: application
    container: {
      image: 'ghcr.io/radius-project/samples/dapr-backend:latest'
      ports: {
        orders: {
          containerPort: 3000
        }
      }
    }
  }
  //connection provides component name, not
connection string
connections: {
  orders: {
    source: stateStore02.id
  }
}
extensions: [
  {
    kind: 'daprSidecar'
    appId: 'backend'
    appPort: 3000
  }
]
}

// The Dapr state store that is connected to the
backend container
resource stateStore02 'Applications.Dapr/stateStores@2023-10-01-preview' = {
  name: 'statestore02'
  properties: {
    // Provision Redis Dapr state store automatically
    via Radius Recipe
    environment: environment
    application: application
    resourceProvisioning: 'recipe'
    recipe: {
      name: 'default'
    }
  }
}

```

The example has two Recipes, one that defines a containerized State Store, and one that defines an Azure CosmosDb State Store. Have a look at the simplified version of a Recipe for dev/test below. It uses Kubernetes resources to deploy Redis, and a Dapr component to describe it.

It returns the Dapr Component so Radius can use it to create Connections.

Recipe 1^[3]

```
extension kubernetes with {
  //Kubernetes extension
} as kubernetes

param context object

resource redis 'apps/Deployment@v1' = {
  //Kubernetes deployment that deploys Redis containers
}

resource svc 'core/Service@v1' = {
  //Kubernetes service that connects to Redis Pods
}

resource daprComponent 'dapr.io/Component@v1alpha1' = {
  //Dapr component that describes the State Store
}

//Mandatory output for Radius to connect State Store to App
output result object = {
  resources: [ .. ]
  values: {
    type: daprType
    version: daprVersion
    metadata: daprComponent.spec.metadata
  }
}
```

In the script below, you can see a simplified version of a Recipe that creates an Azure Cosmos Db State Store. By referencing this template from the Environment, the application will be deployed with a production-ready State Store.

Recipe 2^[4]

```
param context object
param location string = 'northeurope'
param accountName string = context.resource.name
param dbName string = context.resource.name
param appId string

// Cosmos DB Account
resource cosmosDbAccount 'Microsoft.DocumentDB/databaseAccounts@2021-06-15' = {
  //Azure resource spec
}

resource database 'Microsoft.DocumentDB/databaseAccounts/sqlDatabases@2022-05-15' = {
  //Azure resource spec
}

resource container 'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers@2022-05-15' = {
  //Azure resource spec
}
```

```
resource daprComponent 'dapr.io/Component@v1alpha1' = {
  //Dapr component that describes the State Store
}

output result object = {
  resources: [..]
  values: {
    type: daprType
    version: daprVersion
    metadata: daprComponent.spec.metadata
    server: cosmosDbAccount.properties.documentEndpoint
    database: dbName
    collection: containerName
    port: 443
  }
}
```

Bringing it together

You now know that you can define different Environments, each defining the same resource types, with different concrete implementations. On non-production environments you can use containerized datastores, like Redis from Recipe 1. On production environments, you should use PaaS services like Azure Cosmos Db from Recipe 2 above.

This is what the Environment definitions would look like:

Non-production Environment

```
resource env 'Applications.Core/environments@2023-10-01-preview' = {
  name: 'test'
  properties: {
    recipes: {
      //containerized Dapr State store recipe
      'Applications.Dapr/stateStores': {
        default: {
          templateKind: 'bicep'
          templatePath: 'acrradius.azurecr.io/recipes/localstatestore:0.1.2'
        }
      }
    }
  }
}
```

Production Environment

```
resource env 'Applications.Core/environments@2023-10-01-preview' = {
  name: 'prod'
  properties: {
    recipes: {
      //CosmosDb Dapr State store recipe
      'Applications.Dapr/stateStores': {
        default: {
          templateKind: 'bicep'
          templatePath: 'acrradius.azurecr.io/recipes/cosmosstatestore:0.1.0'
        }
      }
    }
  }
}
```

By deploying the **same** Application definition in two **different** environments, it will use different state stores.

³ <https://github.com/loekd/radius-demos/blob/main/02-Dapr/recipes/stateStoreRecipe.bicep>

⁴ https://github.com/loekd/radius-demos/blob/main/02-Dapr/recipes/cosmos_statestore_recipe.bicep

Notice how the resource type of `stateStore02` matches the Recipe's resource type, and that the `stateStore02.properties.recipe.name` ('default') also matches the Recipe's name in the Environment. You can use multiple Recipes for the same resource type, by using different names.

```
resource stateStore02 'Applications.Dapr/stateStores@2023-10-01-preview' = {
  name: 'statestore02'
  properties: {
    // Provision Redis Dapr state store automatically
    via Radius Recipe
    environment: environment
    application: application
    resourceProvisioning: 'recipe'
    recipe: {
      name: 'default'
    }
  }
}
```

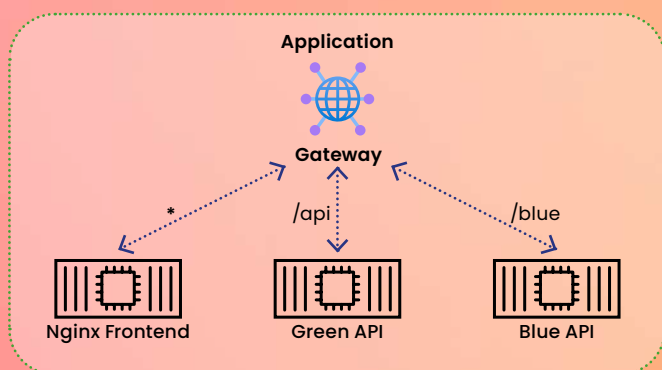
Networking

The last thing we need to arrange is networking. Up until now, we have been using tunneling to access the application through `localhost`. For this, we can use a Gateway.

Gateway

Usually, a Cloud-native app will have a web based interface. Often, it will have an API. Using a Radius Gateway, you can expose both at the same domain, by creating traffic routing rules. It has basic support for path rewriting. You can also use a Gateway for TLS offloading. Below, you can see an example of a Gateway that routes traffic for a host named 'example.com'.

It examines the incoming web request path. If the path contains `/api` or, it will send that request to the container named `green`. If the request path contains `/blue`, the request will be sent to a container named `blue`. Also the matched word `blue` will be stripped from the path sent downstream, because of the `replacePrefix` part. This demonstrates how simple request rewriting works in Radius. All other requests will be sent to the `nginx` container, which runs a web frontend.



```
resource gateway 'Applications.Core/gateways@2023-10-01-preview' = {
  name: 'gateway01'
  properties: {
    application: app.id
    environment: env.id
    internal: true
    hostname: {
      fullyQualifiedHostname: 'test.loekd.com'
    }
    routes: [
      {
        path: '/api'
        destination: 'http://green:8080'
      }
      {
        path: '/blue'
        destination: 'http://blue:8082'
        replacePrefix: '/'
      }
      {
        path: '/'
        destination: 'http://nginx:80'
      }
    ]
  }
}
```

If you want to deploy a working version of a Gateway^[5].

- call blue API: `curl -HHost:test.loekd.com http://localhost/blue/api/color`
- call green API: `curl -HHost:test.loekd.com http://localhost/api/color`
- call the main site: `curl -HHost:test.loekd.com http://localhost`

Conclusion

Radius transforms the development and deployment process by abstracting the complexities of modern Cloud environments. By using Applications, Connections and Recipes, it allows developers to focus on building features rather than managing infrastructure. By defining Recipes, people with an operations role can ensure that that infrastructure is compliant. The Application graph facilitates people in both roles to understand and discuss Applications and their dependencies.

At the time of writing, Radius is still in its early stages. However, especially within larger organizations, and once generally available, Radius could be a valuable tool for software development teams. Make sure to get some hands on experience with it today, provide feedback to the team or contribute code yourself.

This could also mean that the samples in this document are outdated and no longer work. Breaking changes are not uncommon while this platform is being built. The concepts and ideas however, will likely remain the same.

⁵ https://github.com/loekd/radius-demos/blob/main/03-Gateway/app_gw.bicep

Did You Update The Documentation?

Nobody likes writing documentation. It's a boring and tedious task. And the worst thing is that the moment you write it, it's already outdated. Yet, it's one of the most important parts of the software we create. No matter how many times developers say, "The code is self-explanatory," it's not. Documentation helps new people get onboarded more quickly, and more importantly, it can improve collaboration between teams. Documentation is key when you build applications that expose APIs.

Authors Matthijs van der Veer and Rutger Buiteman

In the case of APIs, most developers will first ask for an OpenAPI specification as their documentation of choice. If you're unfamiliar with OpenAPI, it's a standard way to describe your API. This file contains all the information about your API, like the endpoints, the request and response bodies, and the authentication methods. It often surfaces through a Swagger UI, which provides a friendly way to explore the API from your browser. These files are automatically generated from your codebase, so creating and sharing them is not a problem at all. However, not everybody understands these specifications or has the tools to interpret them. Some people, yes, even developers, prefer to read human-readable documentation.

This presents us with a problem. The typical developer stereotype is that we don't like the following three things:

- Writing documentation
- Writing unit tests
- Writing regular expressions

Unsurprisingly, when the newest wave of LLMs hit in 2022, people started making tooling that solved the latter two. Github Copilot, especially, has been a game-changer for many developers. While you can use it to extract documentation from your code, it suffers from one big problem: it's a copilot, not a captain. It can help you write documentation, but you need to tell it to do so.

If you aren't going to remember to write documentation, why would you remember to ask Copilot to write it for you? So, in true developer fashion, the best way to get out of doing something boring is to write a tool for it.

Continuous Integration/ Continuous Documentation

We're currently building an Integration Platform for our client. The platform offers developers a quick way to get their integrations off the ground, no

matter the programming language they want to use. Many of these integrations are APIs that all need to be documented. In our Platform Engineering journey, we spend a lot of time and attention on perfecting our integration and deployment pipelines. Next to all the big jobs like building, testing, and deploying, we use these pipelines to make our jobs easier. So we decided to put the machines to work on doing the task we would surely forget. We can automatically generate human-readable documentation from

the OpenAPI specification for all the APIs that land on our platform.

After we build, test, and deploy our software, we need to acquire the OpenAPI specification and offer it to a Large Language Model (LLM). The software we wrote to do this could have been implemented in various ways, from building a custom pipeline step to using a simple script in our pipeline. Whatever the solution, we need credentials to access the LLM, as well as credentials to push the documentation to Confluence.



While we could securely obtain these credentials in all our pipelines, they would need to be accessible by all the pipelines that we run for these different APIs. Instead, we decided to build a separate service in our integration platform that all the pipelines can call. This service generates the documentation and pushes it to Confluence. All our API pipelines run with their own service principal, which can be granted access to this service. This way, we can keep the credentials for the LLM and Confluence in one place and only have to manage them in one place.

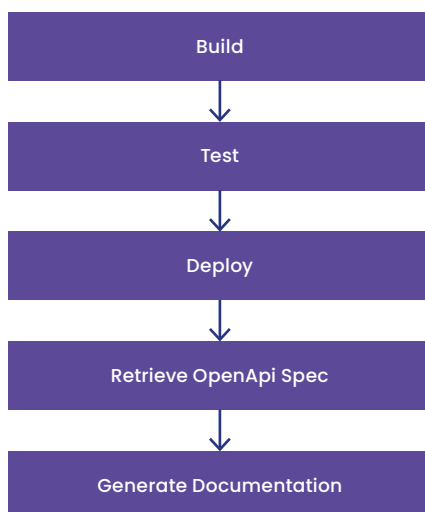


Figure 1: Automatic Documentation Generation

The Service

This central service is an API built on ASP.NET Core and leverages two different technologies: Semantic Kernel and Prompty. While you can achieve the same with the Azure OpenAI REST API, these tools will make your life easier. Semantic Kernel abstracts away specific LLM implementations and is helpful in bootstrapping authentication to Azure OpenAI. Prompty is a convenient way to bundle your prompt, parameters and sample input in a single file.



You can find out more about Prompty in another article in this magazine: "Stop Creating Content With ChatGPT!". Together, these two tools make interacting with LLMs a breeze. It reduces the code you need to execute to only a few lines:

```
KernelArguments kernelArguments = new()
{
    { "specification", openApiSpec }
};

var prompty = kernel.CreateFunctionFromPromptFile
("./Prompts/openapi.prompty");
var promptResult = await prompty.InvokeAsync<string>
(kernel, kernelArguments);
```

The Prompty file includes all the instructions the LLM needs to turn a complex JSON file into English. It also contains part of the Confluence documentation that describes the specific formatting rules for a Confluence page. This way, the LLM can generate the documentation in the correct format. The API then takes the generated documentation and uploads it to Confluence.

All the different workloads that make up our integration platform run on our Azure Landing Zones. These landing zones come with a lot of benefits for networking, security, and governance. To host this central service, we gave it its own Azure Landing Zone. The Landing Zone offers us a blank canvas to deploy our service, and in our case, this is limited to just a few services.

First of all, we need something to run our code. Azure offers many different compute products to host a simple API. We decided on an Azure Container App. This has easily become our hosting model of choice in our integration platform because it offers container-based hosting with many different scaling options. This might sound like overkill for a relatively simple API that is called maybe once or twice per hour, but a benefit is that the API can automatically scale back to zero instances, so it does not cost any money. This makes it a very cost-effective solution.

Next to the Container App, the API needs to interact with an LLM in order to generate the documentation. For this, we decided to use the Azure OpenAI service and deployed the GPT-4o model. Through Azure OpenAI, we get access to all leading LLMs without a matching price tag. Just like with Container Apps, you only pay for what you actually use. In the case of our GPT-4o model, we only pay for the amount of tokens we exchange. When we generate documentation, the LLM costs are below 2 cents per run.

When no documentation is generated, we pay nothing for these resources. The API sets up a secure connection through the Managed Identity, which is built into the Container App, which saves us from having to manage credentials for it. As long as the identity has the correct access permissions, it can connect.

Finally, we also need a place to store the credentials that are needed to upload the documentation to Confluence. This is simply solved by adding a Keyvault and having the API connect to it to retrieve the necessary credentials.

Using it

Now that it is all available and running, we can actually use it in our pipelines. We have multiple environments, and as all of them can have different versions of an API running, we need to generate the documentation per environment. We have accomplished this by adding a 'Generate documentation' step after all our deployments, which will retrieve the OpenAPI specification from the deployed resource and call the documentation service with the retrieved specification as content, together with the identifier of the environment. The documentation service will make sure that a page on Confluence is created or updated for that environment and that the generated documentation is on that page. To make it easier for all users of our platform, we turned it into a templated step. So, all they have to do is add the following to their pipeline:

```
- job: generate_documentation_nonprod
  displayName: 📄 Generate Documentation Nonprod
  steps:
    - script: curl -o swagger.json https://apiname.
      nonprod.integrations.clientname.org/swagger/v1/
      swagger.json
      displayName: Download Swagger JSON
    - template: /pipelines/templates/steps/generate-
      documentation.yml@platform
      parameters:
        name: ApiName
        environment: nonprod
        filePath: swagger.json
```

Conclusion

We're building an integration platform that will host dozens of APIs by different authors using different programming languages. In every decision that we make on our Platform Engineering journey, we try to maximize the automation to make the developer experience the best we can. All of the users of our platform can now use our documentation generation API to make certain that their documentation continuously matches their actual features. This way, we can ensure that the documentation is always up to date and that we can focus on the things we love to do, such as adding value by building features.

A hand with a white glove points upwards against a brick wall. The wall is painted with various colors like orange, red, blue, and purple, giving it a textured, artistic appearance. The hand is wearing a white glove and is pointing towards the top left of the frame.

**Ready for
something
different?
Join us.**

**Let's explore the
opportunities
together, no
strings attached.
Let's meet.**

<https://xebia.com/careers/>

Creating Digital Leaders.



If you prefer the digital
version of this magazine,
please scan the qr-code.

xebia.com

