Model Driven Engineering Project Report

3A GL

Pygram: from diagrams as code to infrastructure as code

Realized By

Yassir Douslimi

Youssef Sirajeddine

Hanane Mouhim

Hicham Talbi

Supervised By

Pr. Mahmoud El Hamlaoui

2020 – 2021

Contents

List of Figures

Chapter 1

Introduction

Introduction

One of the most promising aspects of modern technology is the cloud nature of how we build and deploy applications on a massive scale. The ecosystem around this fact has grown exponentially in the last several years and it's especially interesting to consider the ways we study and conceive these systems because their complexity grew with the numerous services that are shipped.

This project focuses on building a way to transform ideas into reality by converting the code that generates diagrams into full fledged configuration files that can build real life clusters and infrastructure.

Using Jetbrains MPS, we hope to construct a model for a language that describe the design of diagrams relating to cloud architectures and then attempt to transform it into text in the format of terraform scripts.

Keywords:

Model Driven Engineering, Meta Programming System, Python, Diagrams as Code, Terraform, Infrastructure as Code

Chapter 2

Diagrams as Code

2.1 Diagrams

All major cloud providers possess a collection of services and products and most of them organize them through icons that represent the service. This lead to the ability to draw diagrams specific to architectures of systems in the cloud.

Cloud Architecture Diagrams are used to visually record the enterprise cloud computing services provided by a company. Because the infrastructure of these services can be complicated, drawing a cloud architecture diagram for your organization's documentation, making plans for changes, or troubleshooting issues is a good idea.

2.1.1 AWS Architecture Diagrams

The most popular cloud computing platform is Amazon's AWS. AWS provides approximately 200 services in categories such as compute, networking, and storage. The deployment, topology, and design of Amazon Web Service products and resources produced on its cloud platform are described in AWS Architecture Diagrams.

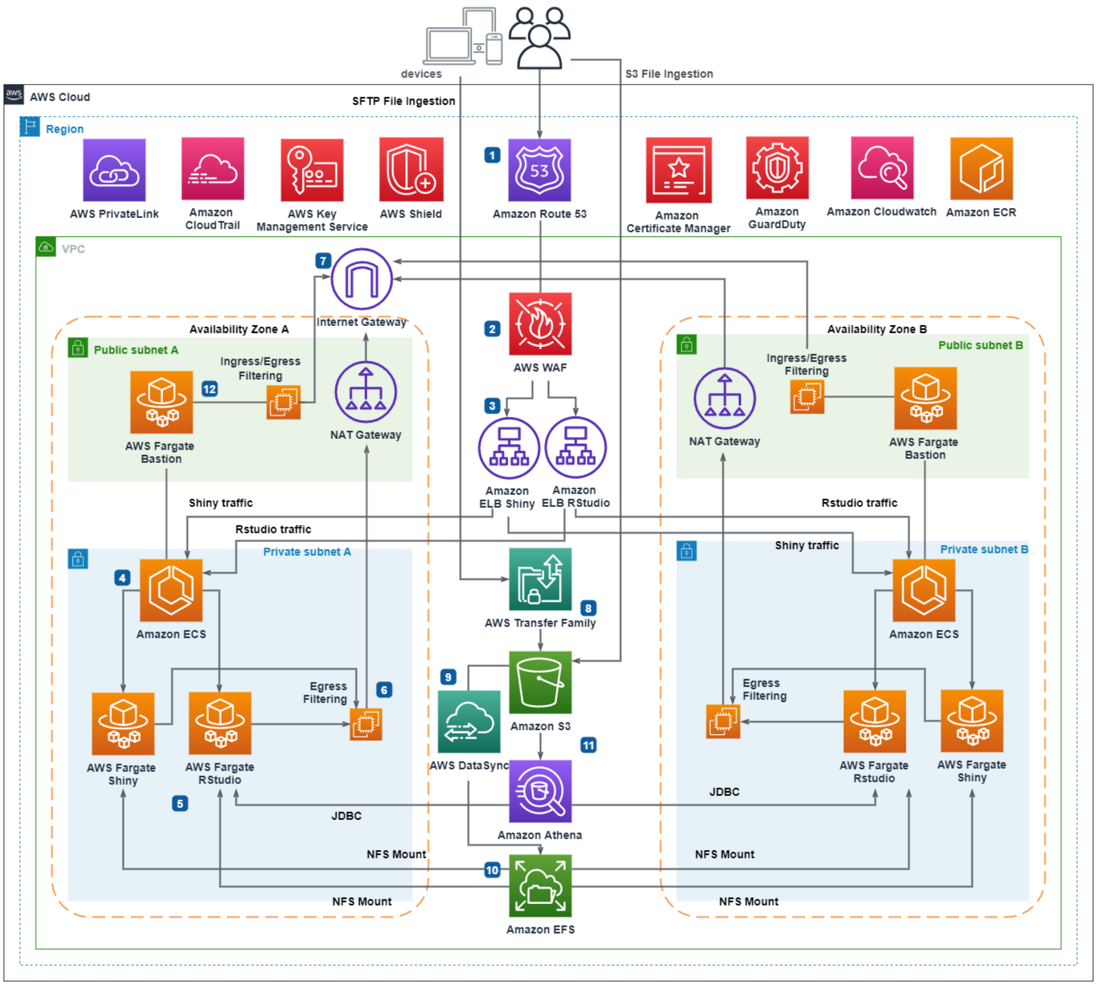


Figure 2.1: AWS Example Architecture

2.1.2 *Azure Architecture Diagrams*

The Microsoft Azure platform is the company's cloud offering. It's the second-most popular cloud computing service, and it's ideal for companies who use Microsoft's other products. As a result, it is the most popular option for businesses on the market.

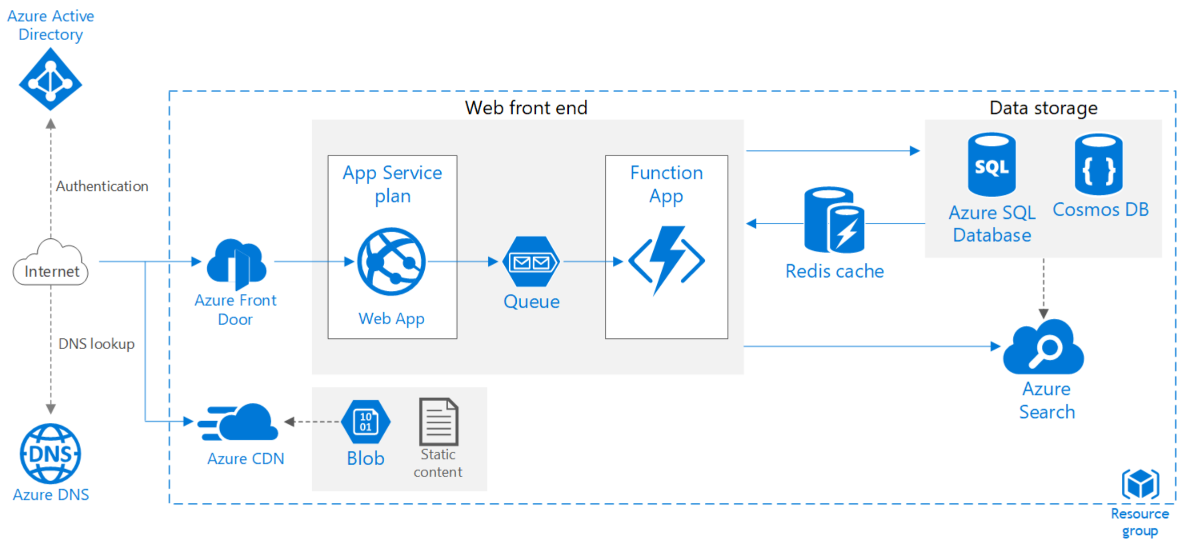


Figure 2.2: Azure Example Architecture

2.1.3 GCP Architecture Diagrams

Google Cloud Platform is the abbreviation for Google Cloud Platform. These cloud computing services were created by Google to support its own products, such as Google Search, Gmail, and Youtube. It is the third-largest Infrastructure-as-a-Service (IaaS) provider and is considered a notable leader in the market according Gartner.

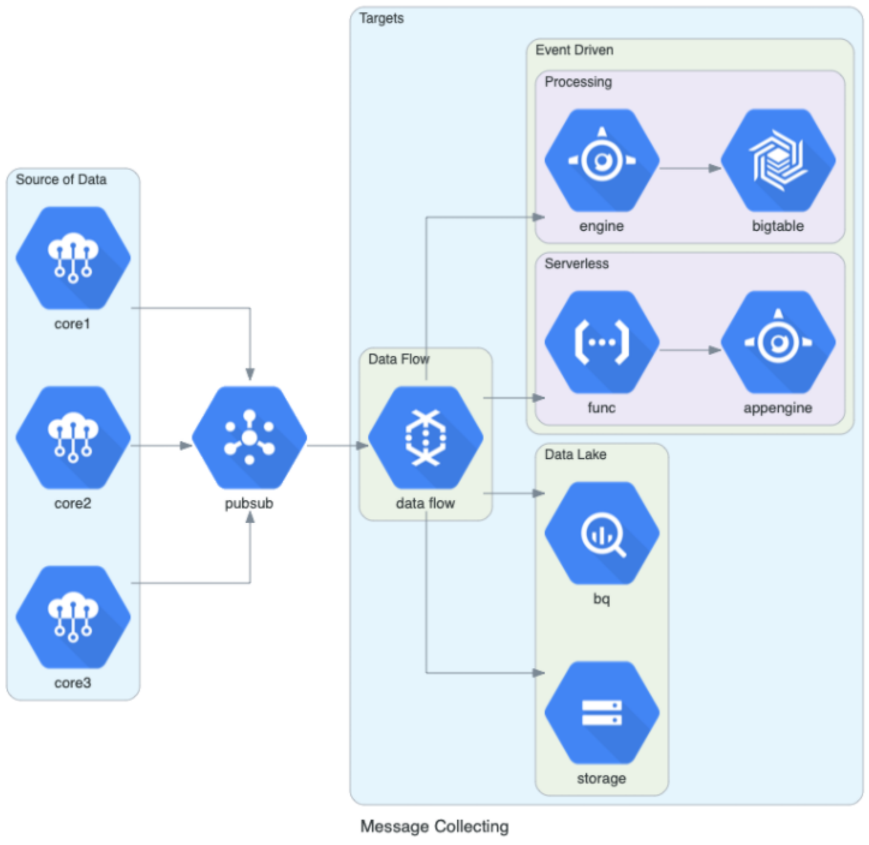


Figure 2.3: GCP Example Architecture

2.1.4 On-Prem Architecture Diagrams

On-premises refers to a software and hardware infrastructure architecture that is deployed and managed within the bounds of your company. The infrastructure setup is completely within your control. Data is kept in your private network, and only members of your team have access to it. On-premises infrastructure is sometimes known as a private cloud.

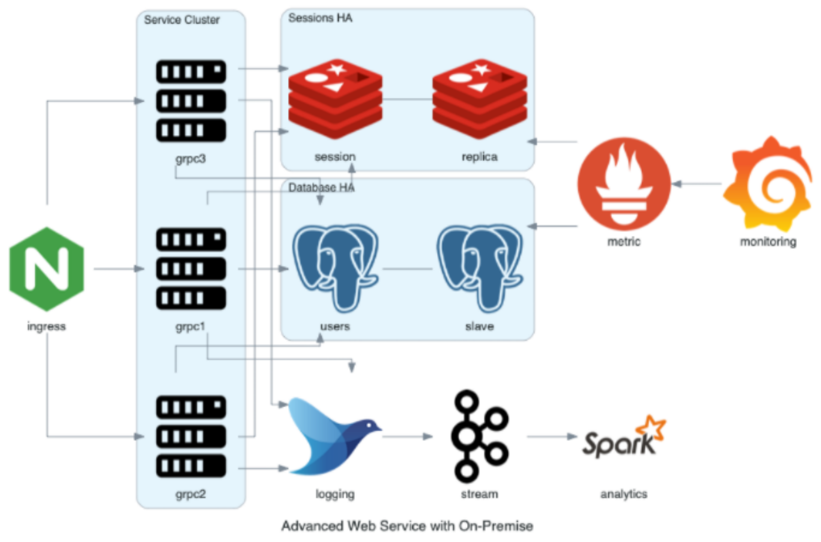


Figure 2.4: On Premise Example Architecture

2.2 Diagrams as Code

In Python programming, you can use mingrammer/diagrams to draw the cloud system architecture. It was created with the intention of developing a novel system architecture design without the use of any design tools. You can also use this tool to describe or visualize the current system architecture. AWS, Azure, GCP, Kubernetes, Alibaba Cloud, Oracle Cloud, and other major providers are now supported by Diagrams. On-premise nodes, SaaS, and major programming frameworks and languages are also supported.

2.2.1 *Sample code*

```
# diagram.py
from diagrams import Diagram
from diagrams.aws.compute import EC2
from diagrams.aws.database import RDS
from diagrams.aws.network import ELB

with Diagram("Web Service", show=False):
    ELB("lb") >> EC2("web") >> RDS("userdb")
```

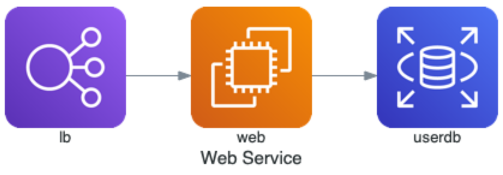


Figure 2.5: Output of code example 1

```
from diagrams import Cluster, Diagram
from diagrams.aws.compute import ECS, EKS, Lambda
from diagrams.aws.database import Redshift
from diagrams.aws.integration import SQS
from diagrams.aws.storage import S3

with Diagram("Event Processing", show=False):
    source = EKS("k8s source")

    with Cluster("Event Flows"):
        with Cluster("Event Workers"):
            workers = [ECS("worker1"),
                       ECS("worker2"),
                       ECS("worker3")]
        queue = SQS("event queue")

        with Cluster("Processing"):
            handlers = [Lambda("proc1"),
                        Lambda("proc2"),
                        Lambda("proc3")]

    store = S3("events store")
    dw = Redshift("analytics")

    source >> workers >> queue >> handlers
    handlers >> store
    handlers >> dw
```

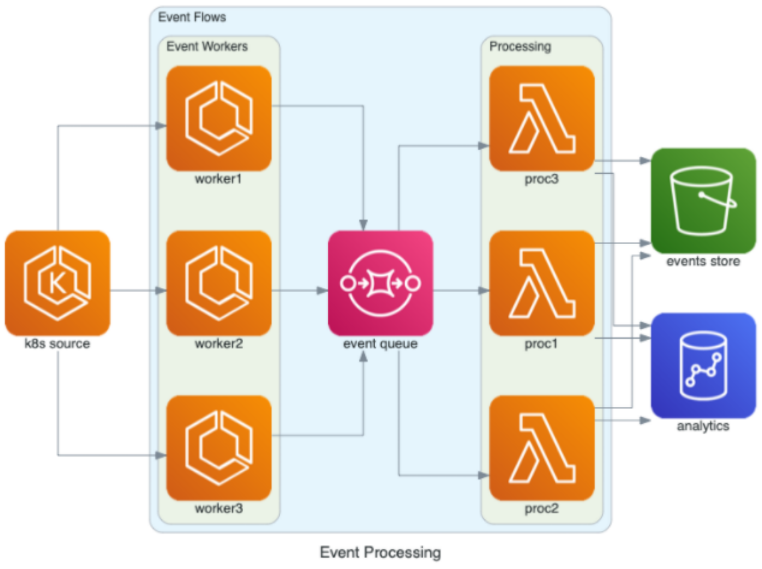


Figure 2.6: Output of code example 2

2.3 Model of Diagrams as Code

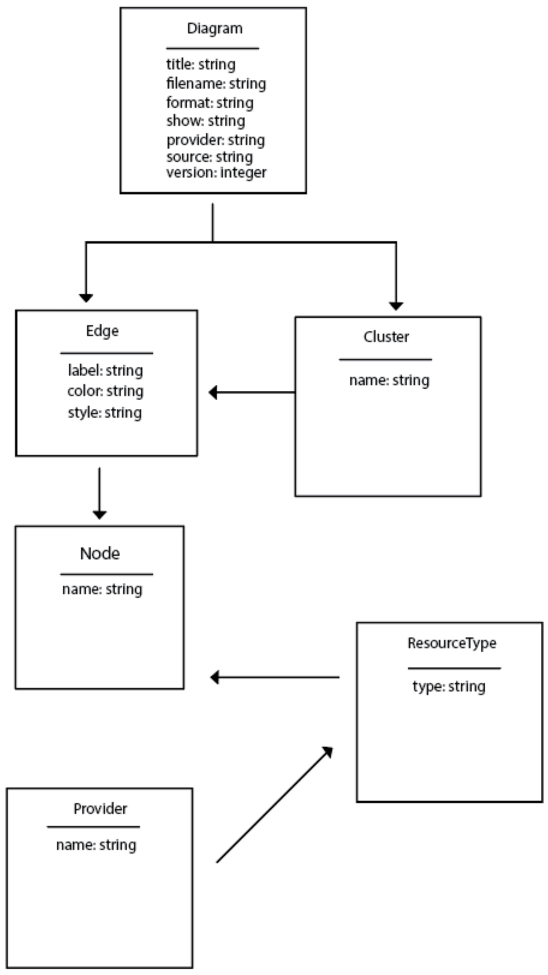


Figure 2.7: Diagrams as Code Model

Chapter 3

Terraform & Infrastructure as Code

3.1 Infrastructure as Code

Infrastructure as Code (IaC) is a descriptive approach for managing infrastructure (networks, virtual machines, load balancers, and connection architecture) that employs the same versioning as the DevOps team does for source code. An IaC model generates the same environment every time it is applied, similar to the principle that the same source code generates the same binary. IaC is a crucial DevOps practice that works in tandem with continuous delivery.

Infrastructure as Code was created to address the issue of release pipeline environment drift. Without IaC, teams are responsible for maintaining the settings of each deployment environment. Each environment evolves into a snowflake, a one-of-a-kind arrangement that cannot be replicated automatically. During deployments, inconsistency among environments causes problems. Snowflakes need manual operations that are difficult to track and contribute to errors in infrastructure administration and maintenance.

Infrastructure as Code adheres to the idea of idempotence. Idempotence refers to the property that a deployment command always configures the target environment in the same way, regardless of the environment's beginning state.

Idempotency is achieved by either automatically setting an existing target or dismissing the existing target and starting over.

As a result, using IaC, teams modify the environment description and version the configuration model, which is often written in well-documented code formats like JSON. The model is run in the release pipeline to setup target environments. The team edits the source, not the target, if they need to make changes.

3.2 Terraform

Figure 3.1: Terraform Logo

HashiCorp invented Terraform, an open-source infrastructure as code software application. Users define and provide data center infrastructure using HashiCorp Configuration Language (HCL) or alternatively JSON, a declarative configuration language.

Terraform works with "providers" to manage external resources (such public cloud infrastructure, private cloud infrastructure, network appliances, software as a service, and platform as a service). HashiCorp has a large number of approved providers and can also work with community-developed providers.

Terraform providers can be interacted with by declaring resources or contacting data sources. Terraform employs declarative configuration to express the intended final state, rather than utilizing imperative instructions to provision resources. Terraform will undertake CRUD operations on behalf of the user to achieve the required state once a user runs Terraform on a particular resource. Code for the infrastructure can indeed be written as modules, allowing for reusability and maintainability.

3.2.1 Terraform File Structure

```
    terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 1.0.4"
    }
  }
}

variable "base_cidr_block" {
  description = "A /16 CIDR range definition that the VPC will use"
  default = "10.1.0.0/16"
}

variable "availability_zones" {
  description = "A list of availability zones in which to create subnets"
  type = list(string)
}

provider "aws" {
  region = var.aws_region
}

resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}
```

Chapter 4

Meta Programming System

4.1 Jetbrains MPS

Figure 4.1: MPS Logo

Subject-specific dialects and languages assist people in communicating clearly and effectively about their respective subjects. In the world of programming languages, MPS gives the same flexibility. Unlike traditional programming languages, which have rigid syntax and semantics, MPS allows users to develop, edit, and extend a language from the ground up.

The MPS editor's job is to display the AST in a user-friendly manner and give tools for properly altering it. If you're designing a language that mimics a regular textual language, for example, the editor should deliver an experience similar to that of using a standard text editor. Similarly, if you're building a language that uses graphical notations, the editor should provide you the same experience as a diagramming editor.

You establish the rules for modifying the code and how the code is presented to the user when you create a language in MPS. You can also declare the type system and restrictions of the language as a set of rules that apply to your language. These features, when combined, allow MPS to check program code on the fly, making programming with the new language simple and error-free.

MPS employs a generative strategy. This implies you can create generators for your language to convert user input into a more standard, often general-purpose language. At the moment, MPS excels at, but is not limited to, creating Java code. C, C, XML, FHTML, PDF, LaTeX, JavaScript, and more can all be generated.

4.2 MPS Project Hierarchy

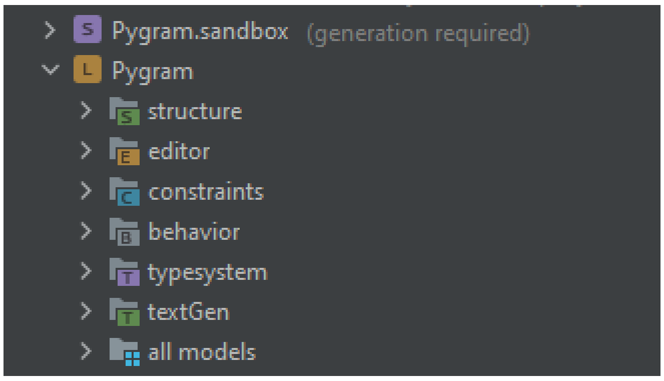


Figure 4.2: Project Hierarchy

The sandbox element (S symbol) represents where the user interacts with the language. The language however is defined under the L symbol

4.2.1 Structure

Defines the types of nodes that can be used in user models (called Concepts). Each concept is represented by a node in the program (model). Concepts define the properties that children, references, and nodes can have. Concepts can implement ConceptInterfaces and extend other Concepts.

4.2.2 Constraints

Beyond the restrictions specified in Structure, restricts the relationships between nodes as well as the permissible values for properties. Constraints are usually defined as:

- the reference's target scope (a collection of allowed nodes a reference can point to)

- A node can be a child, parent, or ancestor of another node in certain scenarios.
- property values that are permitted
- techniques for accessing properties (getters and setters)

4.2.3 Editor

Rather of constructing a parser to convert code from an editable form (such as text) into a tree-like structure that a machine can handle, MPS proposes the concept of a projectional editor, which allows the user to directly edit the AST. Language designers can use the Editor feature to construct a user interface for editing their concept notions.

4.2.4 Behavior

Concepts can specify methods and static methods that can be invoked on nodes in a polymorphic style, just like classes in OOP. As a result, nodes carry behavior in addition to their attributes and relationships.

4.2.5 Typesystem

Type-system rules are required by languages that need to type-check their code. Where the estimated type differs from the expectations, the MPS type-system engine will analyze the rules on-the-fly, calculate types for nodes, and indicate errors. Non-typesystem statements about the model can also be verified using so-called checking rules.

4.2.6 TextGen

The TextGen phase kicks off when the Generator has reached the bottom-line AST representation and translates all nodes in the model into their textual representation and stores the resulting textual source files on disk during code creation.

4.3 Model Definition

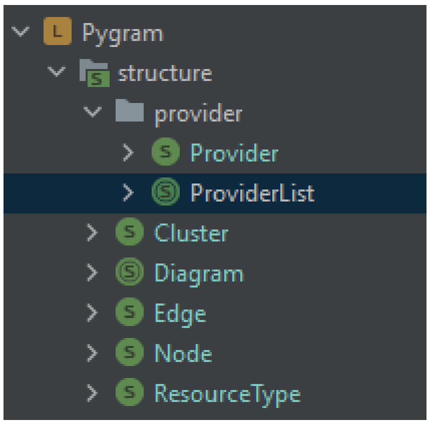
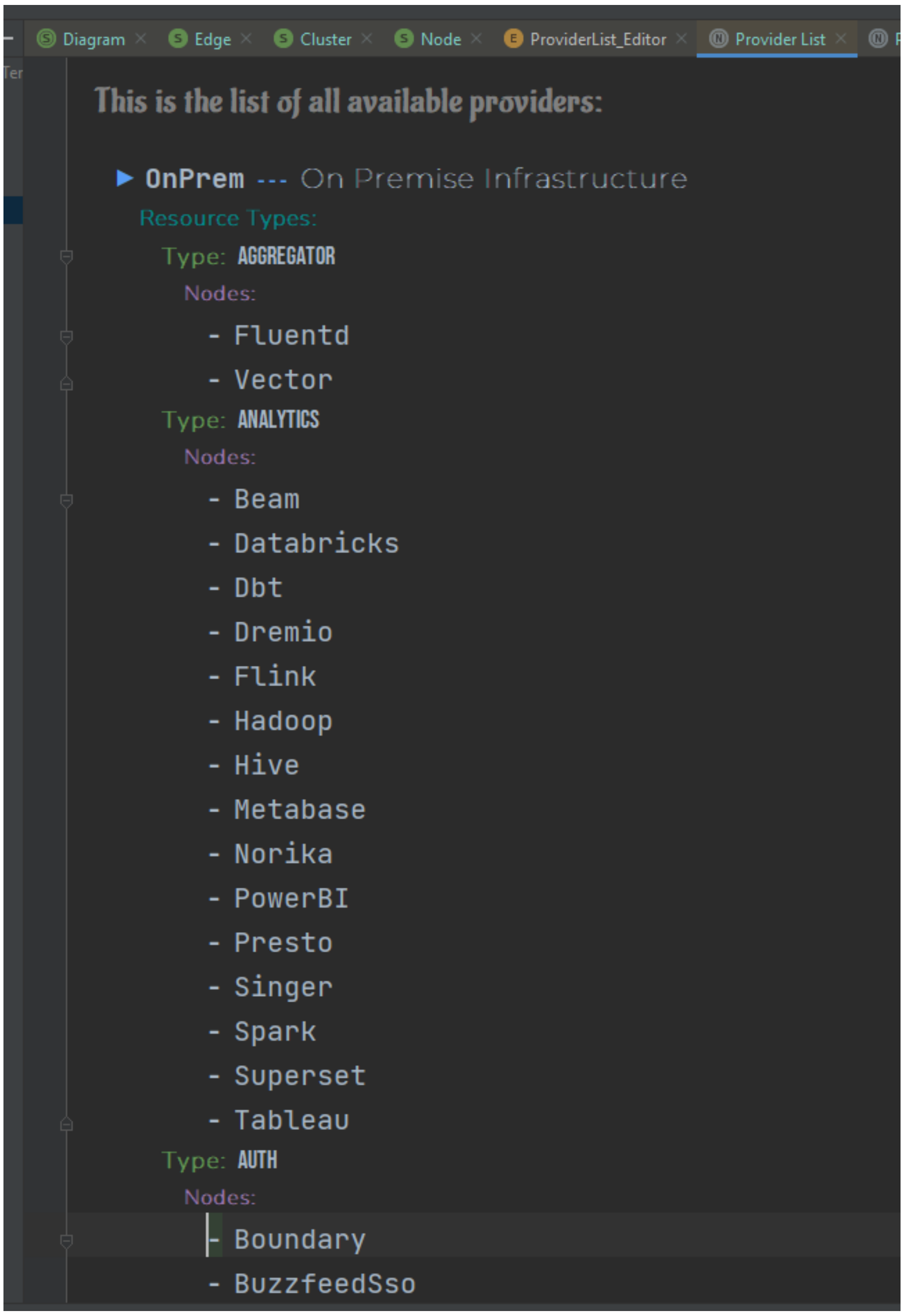


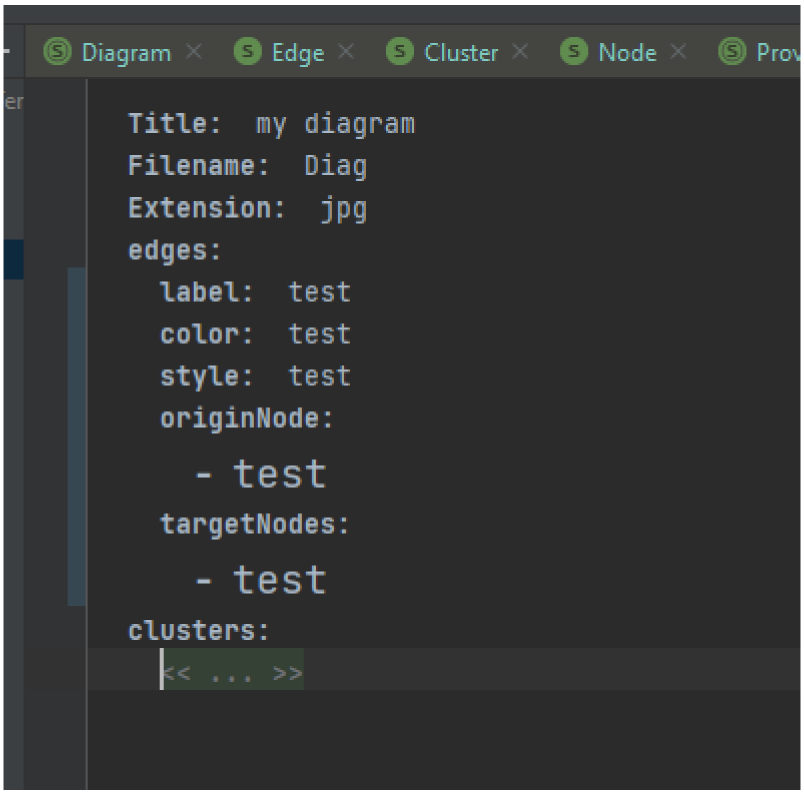
Figure 4.3: The structures of the pygram language



```
This is the list of all available providers:

▶ OnPrem --- On Premise Infrastructure
  Resource Types:
    Type: AGGREGATOR
      Nodes:
        - Fluentd
        - Vector
    Type: ANALYTICS
      Nodes:
        - Beam
        - Databricks
        - Dbt
        - Dremio
        - Flink
        - Hadoop
        - Hive
        - Metabase
        - Norika
        - PowerBI
        - Presto
        - Singer
        - Spark
        - Superset
        - Tableau
    Type: AUTH
      Nodes:
        - Boundary
        - BuzzfeedSso
```

Figure 4.4: The usage of the pygram language through provider lists



```
Title:  my diagram
Filename:  Diag
Extension:  jpg
edges:
  label:  test
  color:  test
  style:  test
  originNode:
    - test
  targetNodes:
    - test
clusters:
  << ... >>
```

Figure 4.5: The usage of the pygram language through diagrams

Chapter 5

Conclusion

As much as the idea of this project was ambitious, the road to fulfilling it was a hard endeavor. Along the journey of trying to find results, we have discovered many concepts and notions about model driven engineering and the conception of domain specific languages that solve specific problems. In tandem with this, we have also deepened our knowledge in cloud technologies notably infrastructure as code and the design of architectures that relate to it.

As much as it was sad to have not reached the full potential of this project, we are nonetheless happy that we started a small step in implementing such an amazing idea of automating the chain of diagrams as code and infrastructure as code with the help of models.

We also want to thank our supervisor, professor Hamlaoui for the chance to discover such an interesting field that has infinite potential. We hope that this project is just the beginning of something big.