Research Project

Software Engineering 2

Marco Molè

AA 2022/2023

Contents

1 Introduction

In this report I'll explore how Kubernetes uses a declarative language to define the desired state of the system. I'll also discuss the limitations of the current state of tools for writing and managing Kubernetes manifests, and propose a solution to address these limitations.

In general terms a declarative language is a programming language that expresses the desired state of the system rather than the steps required to achieve that state. This is in contrast to an imperative language, which expresses the steps required to achieve the desired state.

In the context of Kubernetes, the declarative language is used to define the desired state of the system, which is then used by the Kubernetes control plane to automatically manage the system and ensure that the desired state is achieved and maintained. This approach has several advantages over an imperative approach, including the ability to easily scale the system up or down based on demand, as well as the ability to automatically recover from failures.

2 What is Kubernetes

At its core, Kubernetes operates by organizing containers into logical units called pods, which are then deployed and managed as a single entity. These pods are designed to be highly portable and can be deployed across a range of cloud and on-premises environments, enabling organizations to build and deploy applications in a way that is both scalable and cost-effective.

One of the key benefits of Kubernetes is its ability to automate many of the tasks associated with managing containerized applications, including load balancing, scaling, and failover. This automation not only helps to reduce the operational overhead of managing large-scale container deployments but also ensures that applications are always available and running smoothly.

Overall, Kubernetes represents a significant advancement in the field of container orchestration, providing developers and operations teams with a powerful tool for managing containerized applications at scale. As such, it has become an essential technology for organizations looking to leverage the benefits of containerization and cloud computing to build more agile, scalable, and resilient applications.

The Kubernetes configuration language is expressed in YAML or JSON format, and it consists of a set of declarative statements that describe the desired state of the Kubernetes resources. These statements include specifications for the containers that run within the resources, as well as other settings such as environment variables, ports, and volumes. Being declarative, the configuration language does not specify how the desired state should be achieved, but rather what the desired state is. This allows Kubernetes to automatically manage the resources and ensure that the desired state is achieved and maintained. For example, if a container crashes or becomes unresponsive, Kubernetes will automatically restart it to ensure that the desired state is maintained. Similarly, if a node fails, Kubernetes will automatically reschedule the pods that were running on that node to ensure that the desired state is maintained.

2.1 How Kubernetes works

At its core, Kubernetes is built around the concept of a cluster, which is a group of machines (called nodes) that work together to run containerized applications. The Kubernetes control plane, which is responsible for managing the cluster, consists of several components that work together to provide a robust and scalable platform for deploying and managing containerized applications.

One of the key concepts in Kubernetes is the pod, which is the smallest deployable unit in the system. A pod is a logical host for one or more containers, and it provides a shared environment for those containers to run in. Each pod has a unique IP address and hostname, which makes it easy to communicate with other pods in the same cluster.

The Kubernetes control plane consists of several components, including the API server, etcd, the scheduler, and the controller manager. The API server provides a RESTful API that users can use to interact with the Kubernetes cluster. Etcd is a distributed key-value store that stores the configuration data for the cluster, such as the desired state of the system. The scheduler is responsible for assigning pods to nodes in the cluster based on various criteria, such as resource availability and affinity. Finally, the controller manager is responsible for ensuring that the current state of the cluster matches the desired state.

Kubernetes also provides several key features that make it easy to manage containerized applications, such as service discovery, load balancing, and auto-scaling. Service discovery allows applications to easily find and communicate with other services in the same cluster. Load balancing ensures that traffic is distributed evenly across all instances of a service, while auto-scaling makes it easy to automatically scale the number of instances up or down based on demand.

2.2 Main components

In this section I'll provide a more detailed explanation of the building blocks of Kubernetes, like pods, deployments, services, volumes. This is just a brief overview of the foundational objects of Kubernetes, and it is skipping more complex objects like StatefulSets, DaemonSets, Jobs, Ingress, etc. This is because the focus of this report is on the language used to define the desired state of the system, rather than on the system itself.

2.2.1 Pods

In Kubernetes, a pod is the smallest deployable unit that can be created, scheduled, and managed. Pods can contain one or more containers, which share the same network namespace and can communicate with each other using inter-process communication (IPC). The declarative language of Kubernetes is used to define pods in a Kubernetes manifest file, which is a YAML or JSON file that describes the desired state of the pod.

To define a pod using Kubernetes' declarative language, the manifest file must include the following information:

1. metadata: This includes information such as the name and labels of the pod.
2. spec: This includes information about the desired state of the pod, such as the containers it should contain, the image(s) to use, and the commands to run. In this section are specified the ports to expose, any environment variables to set, and any volumes to mount.

For example, the following YAML manifest file defines a pod with one container:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
spec:
  containers:
  - name: my-container
    image: nginx
    ports:
    - containerPort: 80
```

In this example, the manifest file specifies that the pod should be named “my-pod” and labeled with “app: my-app”. It also specifies that the pod should contain one container named “my-container” that runs the Nginx image and exposes port 80.

Once the manifest file has been created, it can be applied to the Kubernetes cluster using the *kubectl apply* command. Kubernetes will then compare the desired state specified in the manifest file with the actual state of the cluster and make any necessary changes to ensure that the desired state is achieved.

2.2.2 Deployments

A deployment is an object that defines the desired state of a set of replicated pods. Deployments are a higher-level abstraction that enables declarative updates to the desired state of the pod set, allowing for easier management of the underlying resources.

A Kubernetes deployment is responsible for creating and managing a ReplicaSet, which is a Kubernetes object that ensures a specified number of identical replicas of a pod are running at any given time. Deployments provide a way to manage the scaling and rolling updates of pods, by automating the creation and deletion of replica sets as necessary. The deployment object also provides rollback and pause/resume functionality, allowing for more fine-grained control over the lifecycle of the pod set.

When creating a Kubernetes deployment, a user specifies the desired number of replicas, the container images and configurations, and the update strategy. The deployment controller then uses this information to ensure that the desired state of the pod set is achieved and maintained. If the actual state of the pod set deviates from the desired state, the deployment controller automatically performs rolling updates, scaling, or deletion of pods as necessary.

Deployments are a critical component of the Kubernetes platform, as they enable users to manage containerized applications at scale with ease. By abstracting away the complexity of managing individual pods, Kubernetes deployments provide a higher-level interface that is more intuitive for users and reduces the potential for human error. With Kubernetes deployments, users can confidently manage their applications in a highly available and fault-tolerant manner.

To define a deployment using Kubernetes’ declarative language, the manifest file must include the following information:

1. metadata: This includes information such as the name and labels of the deployment.
2. spec: This includes information about the desired state of the deployment, such as the number of replicas, the selector that identifies the pods managed by the deployment, and the configuration of the containers. For example, the following YAML manifest file defines a deployment with 3 replicas:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx
        ports:
        - containerPort: 80
```

In this example, the manifest file specifies that the deployment should be named “my-deployment” and should include three replicas. It also specifies that the deployment should manage pods labeled with “app: my-app” and should create pods with one container running the Nginx image and exposing port 80.

2.2.3 Services

A Service is an abstraction that enables a stable IP address and DNS name to be assigned to a set of Pods. In essence, it serves as a mechanism for defining a logical grouping of Pods and providing a single point of access to them, regardless of the specific Pod that a client may be communicating with at any given time. Services are typically used to enable load balancing and horizontal scaling of application components running within a Kubernetes cluster.

A Service is defined through a declarative YAML configuration file, which specifies the selector label that matches the Pods to be included in the Service, as well as the port(s) that the Service should expose. When a Service is created, Kubernetes automatically assigns it a cluster-unique IP address and creates an associated DNS entry. Requests sent to the Service’s IP address and port are then automatically routed to one of the available Pods that match the Service’s selector, based on a simple round-robin algorithm. This allows clients to access the application components running within a Kubernetes cluster in a consistent and scalable manner, without needing to know the specific IP addresses or hostnames of individual Pods.

Service can be configured in several ways depending on the requirements of the application or workload. Some of the common configurations include:

1. ClusterIP: This is the default configuration for a Service and provides a stable IP address and DNS name within the cluster. The Service is only accessible from within the cluster and is not exposed to the external network.

2. NodePort: This configuration exposes the Service on a static port on each node in the cluster.

This allows the Service to be accessed from outside the cluster using the node's IP address and the static port number.

3. LoadBalancer: This configuration creates a load balancer in a cloud provider's network, which distributes incoming traffic to the Service across multiple nodes. This is typically used when external traffic needs to be distributed across multiple nodes or when a high degree of availability is required.

4. ExternalName: This configuration provides a way to create a Service that simply maps to an external DNS name. This is useful when an application needs to access an external resource, such as a database or web service, using a stable DNS name.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  selector:
    app: mongodb
  ports:
  - protocol: TCP
    port: 27017
    targetPort: 27017
```

The above manifest file defines a Service named "mongo-service" that exposes port 27017 and routes traffic to Pods labeled with "app: mongodb". Other resources within the cluster can then access the Service using the DNS name "mongo-service" and port 27017.

In addition to these basic configurations, Services can also be combined with other Kubernetes features such as selectors, labels, and endpoints to provide more advanced functionality. For example, Services can be used in conjunction with Ingress controllers to provide an HTTP(S) load balancer that routes traffic based on the URL path or hostname of incoming requests.

2.2.4 Volumes

A volume is a directory accessible to containers in a Pod. A volume can be used to store data that needs to be shared between containers, or to persist data beyond the lifetime of a container.

Several types of volumes can be used in Kubernetes, including hostPath volumes, emptyDir volumes, configMap volumes, secret volumes, and persistent volumes. HostPath volumes mount a directory from the host into the container, while emptyDir volumes are created when a Pod is scheduled on a node and deleted when the Pod is terminated. *ConfigMap* volumes and *Secret* volumes provide a way to inject configuration data or sensitive information into a container, respectively. Persistent volumes are used to store data beyond the lifetime of a Pod and can be dynamically provisioned or statically created by an administrator.

To define a volume in Kubernetes, the YAML configuration file would include a volumes section with the desired volume type and configuration options. For example, to define a hostPath volume that mounts the /data directory from the host into a container, the YAML file would look like:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    volumeMounts:
    - name: my-volume
      mountPath: /data
  volumes:
  - name: my-volume
    hostPath:
      path: /data
```

This configuration file specifies that a hostPath volume named *my-volume* should be mounted at */data* inside the container. The *volumeMounts* field in the container specification specifies which volume to mount, and where to mount it.

3 Lack of checks on the configuration of the containers

When writing the specification of a Pod one can specify the container images to run and from which registry pull it from. Container registries are software repositories that store and distribute container images. A container registry is typically used by developers and DevOps teams to store and manage container images, which can be easily shared across different environments, such as development, staging, and production. Container registries provide features like versioning, access control, and image scanning to ensure that container images are secure, reliable, and up-to-date.

Most often, images require configuration through the use of environment variables. The documentation of these variables is generally found on the registry itself, to be consulted by the developer when writing the specification of the Pod.

Configuration errors could be caused by human error, either neglecting to read the documentation, misinterpreting it or by the lack of documentation of the image itself. Misconfiguration could lead to the application not working as expected, or not working at all. It also could be a security risk, since a misconfiguration of security parameters could lead to the use of default credentials, or the use of a not secure protocol.

An erroneous configuration of a container is hard to debug, since the point of failure may not be immediately recognizable due to the high level of coupling of microservices architecture. There is also an additional element of complexity because the application is running inside a virtualized environment, and the failure could be caused by the runtime environment itself.

The manifestations of these errors are mainly two: the application does not work as expected, or it does not work at all. The first case happens when the misconfiguration is not critical, and the application is still able to run, but with some functionalities not working as expected. For example, the application could be running with some default configuration that does not expose all the functionalities of the application.

The second case happens when the misconfiguration is critical, and the application is not able to run at all. For example, no credentials are provided to the application, and the application is not able to connect to other services.

In both cases, the developer has to debug the application to find the cause of the error, and then fix it. This is a time consuming process since the developer has to find the cause of the error, and then fix it.

It could be also a potentially costly error since most K8s clusters exist on a pay-per-use cloud environment.

There is no automatic mechanism that informs the developer if the configuration they have written is correct, or at least if it satisfies some minimum configuration requirements.

4 Proposed automatic mechanism to check the configuration of the containers

The solution I propose is to provide a mechanism that allows the developer to check if the configuration they have written satisfies some minimum configuration requirements, defined by the maintainer of the image. This minimum configuration schema is stored in the registry and is accessible by the developer's IDE when writing the specification of the Pod. The developer can then check if the configuration they have written satisfies the minimum configuration requirements, and if not, the IDE will inform the developer of the missing configuration options and the type of configuration options.

The mechanism I propose leverages the widespread adoption of the Language Server Protocol (LSP)[1] by the most popular IDEs. LSP is a communication protocol that enables the integration of programming language analyzers, such as code editors, with language servers that provide language-specific features such as code completion, error detection, and refactoring. The protocol standardizes the exchange of information between the language server and the client, allowing for interoperability between different code editors and programming languages. This can help developers work more efficiently by providing consistent, language-specific tools across different editing environments.

Firstly I'm going to describe how the configuration schema is defined, then I'm going to describe how the IDE, LSP and registry interact to provide the developer with the configuration schema. In the end, I'll make some considerations on the automatic generation of the configuration schema.

4.1 Schema of the configuration options

The schema of the configuration options is a JSON object that specifies the configuration options of the image, and the type of each configuration option. The schema should contain information about the mandatory configuration options, and the facultative configuration options. The schema should also contain information about the default value of the facultative configuration options.

The format of the schema could be generalized to this structure:

```
{
  "image" : "name of the image:version",
```

[1]https://microsoft.github.io/language-server-protocol/

```
  "example_of_mandatory_configuration_option" : {
    "mandatory" : true,
    "type":"type of the configuration option",
  },
  "example_of_facultative_configuration_option" : {
    "mandatory" : false,
    "type":"type of the configuration option",
    "default": "default value of the configuration option"
  },
}
```

The type of configuration option could be one of the following: string, number, path (ie a string that represents a path), boolean etc. More types definitions could be added if needed.

An interesting case is the case of the default value of a facultative configuration option. The default value could be a string, or it could be the value of another configuration option. In the latter case, the value of the default value should be a string that represents the name of the configuration option, preceded by the *$* symbol. The *$* symbol is used to express that the default value is the value of another configuration option.

This could be useful in the case of a configuration option that is facultative, but that has a default value that depends on the value of another configuration option. This is the case of the configuration options of the postgres image, where the default value of the POSTGRES_DB configuration option is the value of the POSTGRES_USER configuration option.

```
{
  "image": "postgres:latest",
  "POSTGRES_PASSWORD" : {
    "mandatory" : true,
    "type":"string"
  },
  "POSTGRES_USER" : {
    "mandatory" : false,
    "type":"string",
    "default": "postgres"
  },
  "POSTGRES_DB" : {
    "mandatory" : false,
    "type":"string",
    "default": "$POSTGRES_USER"
  }
}
```

Another case is when the default value of a facultative configuration option is a random and auto-generated value. In this case, the value of the default value is written as “!random”. An example of this is the DOCKER_INFLUXDB_INIT_ADMIN_TOKEN variable, which can be found in the example section.

Table 1: Special meaning of the default value of a facultative configuration option

Default value	Meaning
$configuration_option	the default value is the value of configuration_option
!random	the default value is a random and auto-generated value

4.1.1 Another proposal for the notion of type in this context using regular expressions

An alternative to strictly defining the type of the configuration option is to define a regular expression that the value of the configuration option should satisfy. This could be useful in the case of a configuration option that is a string, but that has to satisfy some constraints.

This can be extended to other types, such as numbers, floats, booleans, domains, ipv4, ipv6, urls etc. The following table shows some examples of types and their corresponding regular expressions.

Table 2: Some examples of types and their corresponding regular expressions

Type	regex
string	.*
number	[0-9]+
float	[0-9]+.[0-9]+
unix path	/([A-z0-9/-_]+)/$
boolean	true|false
domain	^([a-z0-9]+(-[a-z0-9]+)*.)+[a-z]{2,}$
ipv4	^([0-9]{1,3}.){3}[0-9]{1,3}$
ipv6	^([0-9a-f]{1,4}:){7}([0-9a-f]){1,4}$
URL	[-a-zA-Z0-9@:%_+.~#?&//=]{2,256}.[a-z]{2,4}(/[-a-zA-Z0-9@:%_+.~#?&//=]*)?

This approach is more flexible than the previous one since it allows us to define the type of configuration option in a more precise way. However, it is also more complex, since it requires the definition of a regular expression for each type of configuration option. This could be a problem in the case of complex types, like the URL type. In this case, the regular expression is very complex, and it is not easy to understand what it does.

Some cases where this approach could be useful are enforcing password complexity, or enforcing the format of a path.

If the regex approach is used, the schema of the configuration options could be generalized to this structure:

```
{
  "image" : "name of the image:version",
  "example_of_mandatory_configuration_option" : {
    "mandatory" : true,
    "regex":"regular expression that the value  should satisfy",
  },
  "example_of_facultative_configuration_option" : {
    "mandatory" : false,
    "regex":"regular expression that the value should satisfy",
    "default": "default value of the configuration option"
  },
}
```

Either the type approach or the regex approach could be used. The type approach is simpler, but it is also less flexible. In my research, I couldn't find any example of an image that requires a complex type that requires a regular expression to be defined. For this reason, I included both approaches in this document, but I will use the type approach in the rest of the document. I think nevertheless that the regex approach could be useful in some cases, like the ones mentioned above.

4.2 Protocol for the communication between the IDE and the registry

The registries should offer an HTTP endpoint that replies with the configuration schema of the requested image, to be consumed by the IDE.

The flow of the interaction between the parties is shown in the following sequence diagram.

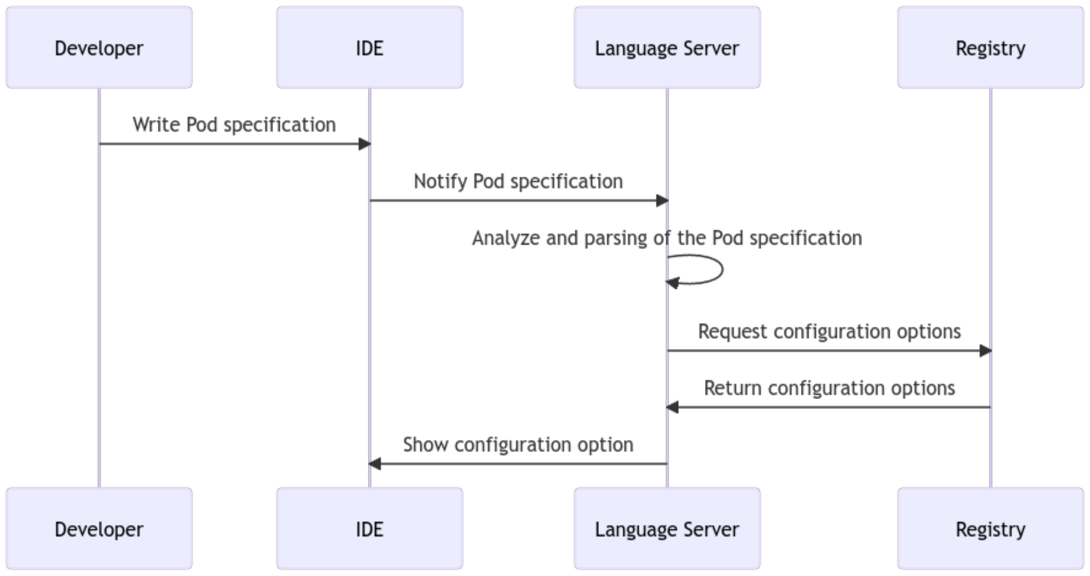
Developer
IDE
Language Server
Registry
Write Pod specification
Notify Pod specification
Analyze and parsing of the Pod specification
Request configuration options
Return configuration options
Show configuration option
Developer
IDE
Language Server
Registry

Figure 1: Sequence Diagram of the interaction between the parties

The developer writes the specification of the Pod in a YAML file, and the IDE sends a request to the registry to retrieve the configuration options of the specified image.

The overhead in traffic and computation is minimal since the registry has to reply to the request of the Language Server with the configuration schema of the requested image, and this happens only when the manifest of the Pod is being written by the developer.

This information is then checked against the PodSpec.containers.env object, which is the object where all the environment variables of the specific containers are specified. The results of this analysis are then shown to the user as warnings or error marks in the IDE.

Recall how Kubernetes objects are structured. The PodSpec.containers.env object is a list of key-value pairs. The key is the name of the environment variable, and the value is the value of the environment variable. Environment variables must be defined as a string.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:  # PodSpec
  containers: # PodSpec.containers
  - name: nginx
    image: nginx
    env: # PodSpec.containers.env
    - name: ENVIRONMENT
      value: "production"
    - name: LOG_LEVEL
      value: "INFO"
```

4.2.1 API for the communication between the IDE and the registry

The registries should offer an HTTP endpoint that replies with the configuration schema of the requested image.

The specification of the API should fit in the already existing API of the registries, I'm taking Docker Hub as an example[2]. The API should be implemented as a new endpoint of the registry API, that is:

- GET /v2/ <name> / schema
  - name: name of the image

The response of the API should be a JSON object that contains the configuration schema of the image as described in the previous section.

4.2.2 How the Language Server checks the configuration options

The Language Server Protocol specification[3] provides a set of methods that the IDE can use to communicate with the Language Server. The methods that are used in this context are:

- initialize: The initialize request is sent as the first request from the client to the server.

[2] https://docs.docker.com/registry/spec/api/
[3] https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

- initialized: The initialized notification is sent from the client to the server after the client received the result of the initialize request but before the client is sending any other request or notification to the server.
- textDocument/didOpen: The document open notification is sent from the client to the server to signal newly opened text documents.
- textDocument/didChange: The document change notification is sent from the client to the server to signal changes to a text document.
- textDocument/publishDiagnostics: Diagnostics notifications are sent from the server to the client to signal the results of validation runs.

The language server when receives the initialize request from the IDE, sends back the initialized notification. The IDE then sends the textDocument/didOpen request to the language server, which contains the content of the YAML file. The language server then parses the YAML file, and extracts the image name and tag. The language server then sends a request to the registry to retrieve the configuration schema of the specified image. The registry replies with the configuration schema of the image, and the language server then checks the configuration options of the YAML file against the configuration schema of the image. The results of this analysis are then sent to the IDE as a textDocument/publishDiagnostics notification.

The same procedure is followed when the user changes the content of the YAML file, which is notified by the IDE sending a textDocument/didChange request to the language server, which contains the changes applied to the YAML file.

For determining on which registry the image is stored, the language server uses the same criteria as the docker and kubernetes clients. The language server first checks if the image name contains a registry address. If it does then it uses that registry. If it does not, then it assumes that the image is stored in Docker Hub.

4.3 Generation of the configuration schema

The current status of documentation of the registries is that it is not machine readable. The documentation is written in a human readable format, and it is not possible to automatically generate the configuration file from the documentation. There is also a lack of standardization of the documentation format, since each registry has its own format, and there is no standard way to retrieve the documentation of an image. Sometimes the documentation is in the image description of the registry, sometimes it's on the website of the application, sometimes it is in the README of the image source code repository.

A possible way to ease the generation of the configuration schema is to ask the image maintainer to provide the configuration schema when they push the image to the registry. This could be done by providing a standard way to specify the configuration schema in the image description of the registry.

I've only looked at Docker Hub, being one of the most popular public registries, that has also searching and filtering capabilities. Docker Hub allows the user to search for images by name and tags. The description of the image is just a free text field.

5 Examples

These are examples based on some of the most popular images taken from the public registry of Docker[4]. These examples were chosen because they were in the most pulled images list of Docker Hub and they have a configuration schema that is neither empty nor too big.

I've chosen databases because they are one of the most used applications in the industry, and they also exhibit a wide range of configuration options.

5.1 mongoDB

MongoDB is a document-oriented database program. It is one of the most popular NoSQL databases. From the documentation[5] of the image, we can see that the image has two environment variables that are mandatory, and they are used to set the username and password of the root user of the database.

```
{
  "image": "mongo:latest",
  "MONGO_INITDB_ROOT_USERNAME" : {
    "mandatory" : true,
    "type":"string"
  },
  "MONGO_INITDB_ROOT_PASSWORD" : {
    "mandatory" : true,
    "type":"string"
  }
}
```

5.2 nginx

Nginx is a web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. From the documentation[6] of the image, we can see that the image has three environment variables that are not mandatory, and they are used to set the template directory, the template suffix and the output directory of the nginx configuration files.

```
{
  "image": "nginx:latest",
  "NGINX_ENVSUBST_TEMPLATE_DIR " : {
    "mandatory" : false,
    "type":"path",
    "default" : "/etc/nginx/templates"
  },
  "NGINX_ENVSUBST_TEMPLATE_SUFFIX " : {
    "mandatory" : false,
    "type":"string",
    "default": ".template"
```

[4]https://hub.docker.com
[5]https://hub.docker.com/_/mongo
[6]https://hub.docker.com/_/nginx

```
  },
  "NGINX_ENVSUBST_OUTPUT_DIR " : {
    "mandatory" : false,
    "type":"path",
    "default": "/etc/nginx/conf.d"
  },

}
```

5.3 InfluxDB

InfluxDB is a time series database. It is optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics.

From the documentation[7] of the image, we can see that the image has five environment variables that are mandatory, and they are used to set the username, password, organization, bucket and retention period of the database. It also has two environment variables that are not mandatory, and they are used to set the admin token and the retention period of the database.

In this example we can see the use of the `!random` keyword, which is used to generate a random string, in this case for the admin token.

```
{
  "image": "influxDB:latest",
  "DOCKER_INFLUXDB_INIT_USERNAME " : {
    "mandatory" : true,
    "type":"string",

  },
  "DOCKER_INFLUXDB_INIT_PASSWORD " : {
    "mandatory" : true,
    "type":"string",

  },
  "DOCKER_INFLUXDB_INIT_ORG " : {
    "mandatory" : true,
    "type":"string",
  },
  "DOCKER_INFLUXDB_INIT_BUCKET " : {
    "mandatory" : true,
    "type":"string",
  },
   "DOCKER_INFLUXDB_INIT_RETENTION " : {
    "mandatory" : false,
    "type":"int",
```

[7]https://hub.docker.com/_/influxdb

```
    "default": "-1"
  },
   "DOCKER_INFLUXDB_INIT_ADMIN_TOKEN " : {
    "mandatory" : false,
    "type":"string",
    "default": "!random" // random string
  },

}
```

5.4 postgres

PostgreSQL is a free and open-source relational database management system emphasizing extensibility and SQL compliance.

From the documentation[8] of the image, we can see that the image has three environment variables, one of which is mandatory, and they are used to set the password, username and database name of the database. In this example we can also see that the default value of the POSTGRES_DB variable is the value of the POSTGRES_USER variable. This is a common pattern in the configuration of images, where the default value of a variable is the value of another variable.

```
{
  "image": "postgres:latest",
  "POSTGRES_PASSWORD" : {
    "mandatory" : true,
    "type":"string"
  },
  "POSTGRES_USER" : {
    "mandatory" : false,
    "type":"string",
    "default": "postgres"
  },
  "POSTGRES_DB" : {
    "mandatory" : false,
    "type":"string",
    "default": "$POSTGRES_USER"
  }
}
```

6 Conclusion

We have seen how a misconfiguration of environment variables could lead to difficult to debug crashes or misbehaviours of the application. In this report I've proposed a solution to the problem of the lack of type checking of environment variables in the declarative language of Kubernetes. The solution is based on the Language Server Protocol, which is a protocol that is used by most of the popular IDEs. The

[8]https://hub.docker.com/_/postgres

solution is based on the idea of generating the configuration schema of an image from the documentation of the image. The configuration schema is then stored on the registry, and it is retrieved by the language server when the user is writing the YAML file. The language server then checks the configuration options of the YAML file against the configuration schema of the image, and it notifies the user of any errors or warnings.

The advantages of implementing this "type checking" mechanism in the the development environment are:

1. Highly portable to multiple types of development environment, due to the high adoption of the LSP protocol.
2. Easily extendible with more code analysis features and eventually adaptable to future changes in the language. A possible code analysis feature could be the check of existence of Pods with a certain label when defining a Service.
3. Flexible, since the declarative language is only one of the ways to configure kubernetes objects (Secrets and ConfigMap can be stored on external providers). The LSP server could be configured to analyze codebases that rely on third party External Secret Store providers.

In the last decade we have seen the rise of a new figure in the software engineering space: the DevOps engineer. Its role is to implement and maintain the ever more complex systems development life cycle and deployment. The one who deploys the software is not the one who wrote the application. This solution could facilitate the knowledge exchange between the two parties and reduce the possibility of errors when deploying a big system with a lot of different components.