Maestro Cloud Control Architecture Overview

Developer's Guide

M3DG-07

February 2026

Version .2.4

Contents

PREFACE

ABOUT THIS GUIDE

The guide describes the main architecture components, approaches and tools, used by Maestro application.

AUDIENCE

The guide is targeted on the developers and architects, working with Maestro application.

THE STRUCTURE OF THE GUIDE

The guide consists of the following sections:

1. Introduction to Maestro – The section provides the general overview of the solution, its main capabilities, delivery models, integrations and permissions.
2. Maestro Components – The section provides the high-level overview of Maestro components
3. Maestro Framework – the section provides the description of Maestro framework, its components, related processes
4. Private Agent – the section provides the details for Maestro private agent, which enables Maestro work with OpenStak and VMware-based private regions.
5. On-Premise Solution – the section describes the on-premise installation of Maestro.
6. Cloud Abstraction Layer – the section covers the main principles of integration with different cloud providers.
7. Event-Driven Architecture – the section descrbes the events-driven approach used within Maestro and events audit principles.
8. M3 SDK – the section describes the main points of M3 SDK configuration and algorhythms.
9. Dynamic UI – the section decribes the specifcis of Maestro UI building, and the respective tools
10. Maestro Databases – the section describes the databases used within Maestro, and the specifcis of MongoDB usage.
11. Support and DevOps tools – the section describes tools and approaches used for Maestro support and DevOps operations.

1 INTRODUCTION TO MAESTRO

Maestro application – is a fault-tolerant system for managing distributed hyper-converged virtual infrastructures. The system is based on event-driven architecture leveraging AMQP protocol (Advanced Message Queuing Protocol), powered by RabbitMQ software application for working with message queuing.

The system contains a private agent component, which is based on a cloud abstraction level. Due to this, the differences, derived from specifics of rendering services by various cloud providers, are hidden.

The cloud abstraction layer allows to manage different virtualizer types via the same code:

- OpenStack (Nova-API)
- CloudStack (CloudStack API)
- Huawei (Open API)
- VMware (vCloudDirector)
- VMware (VSphere)

The application is designed to work in a geographically distributed system, under high load and high availability (99.999%) requirements.

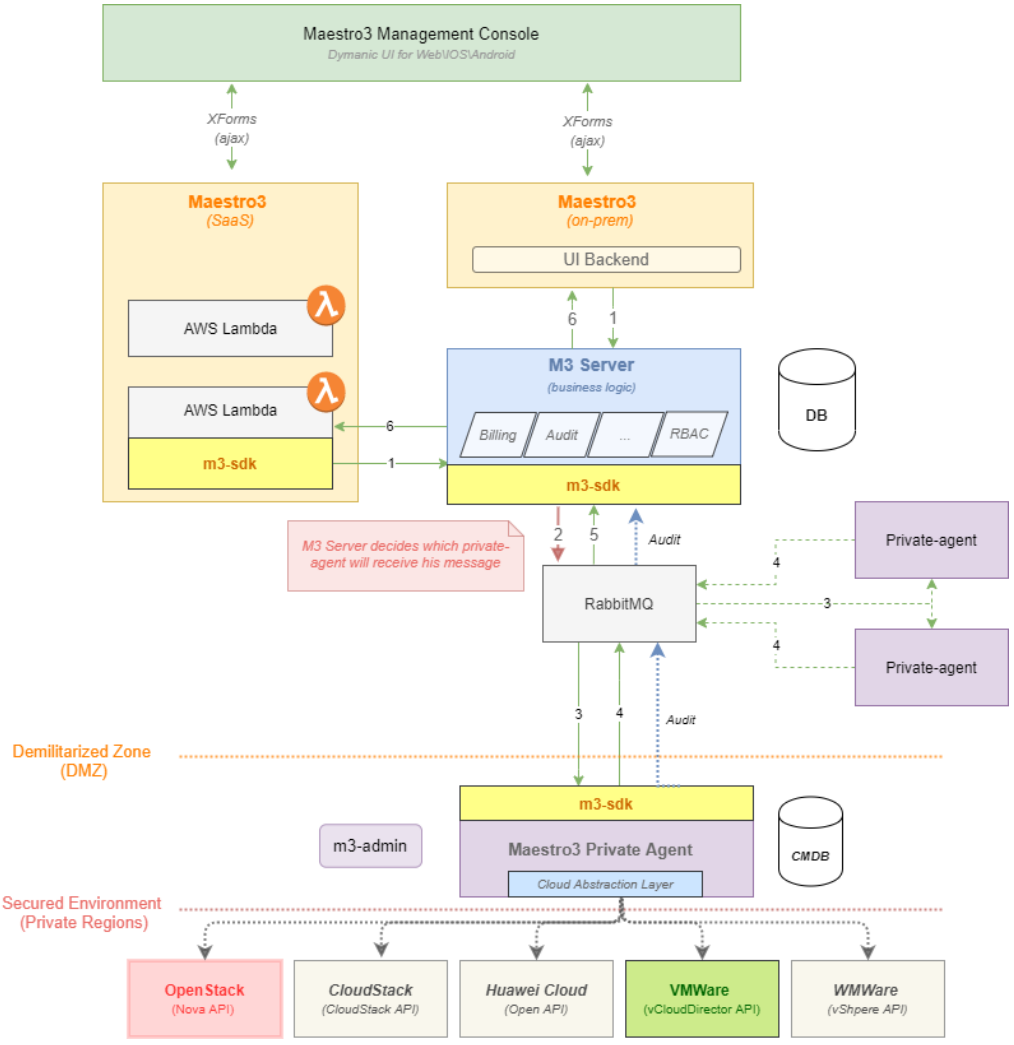


Figure 1 – Maestro architecture framework

All system components are the software manufacturer's in-house development, except for:

- RabbitMQ
- MongoDB (used as a "database" component)
- API of supported public virtual cloud providers and private cloud creation platforms

The main external service that ensures the work of the Maestro application – AWS Lambda.

1.1 MAESTRO MAIN FEATURES AND CAPABILITIES

Maestro is a Cloud Management solution which enables effective, unified, and controllable self-service access to hybrid virtual infrastructures, based on both public and private clouds.

The solution provides users with role-based access to both web and native mobile applications (for Android and Apple). It includes a wide range of events audit, optimization, costs, and billing reports to enable high transparency, accountability, and pro-active optimization for your resources in Cloud.

Maestro purpose and functionality is based on the three pillars, each allowing to establish a high level of virtual infrastructures provisioning, monitoring, and audit:

- **Orchestration, provisioning, brokerage**: Maestro provides a single entry point for creating, reviewing, and managing virtual resources hosted at one or several Cloud providers, both public or private, in a unified way.
- **Costs visibility, audit, reduction**: Maestro includes a set of tools enabling costs visibility, review and control in a convenient unified way for all supported cloud providers.
- **Governance, risk management, compliance (GRC):** With access control, inventory, monitoring, automation and other tools, Maestro is able to effectively support your solutions for modern enterprise-level demands and challenges in GRC.

Maestro has several interfaces for communicating with different types of users:

- **Maestro User Interface** – a web and mobile application providing users with access to Maestro capabilities.
- **Maestro SDK** – a programming interface to enable automation and integration with Maestro user capabilities.
- **Maestro Admin Interface** – a CLI tool available only to Maestro administrative users and allowing Maestro configuration (clouds, tenants, users, etc).

1.2 DEPLOYMENT MODELS

Maestro application can be provided in one of three configurations (deployment models), each of them contains a certain set of functionalities for the end user.

- **The Standard Deployment** model allows users to get access to public virtual cloud providers (AWS, Microsoft Azure, Google Cloud Platform). Available functions are:
    - Unified and simply organized reporting for all customer's resources across all public clouds they use
    - A set of analytics tools for all virtual resources under the customer's account
    - Quotas management tool, that allows to set up the monthly expense limits for virtual infrastructures

  - Alerts and notifications that will inform the customer on the significant events on their resources
- **The Professional Deployment model** allows users to get access to public virtual cloud providers (AWS, Microsoft Azure, Google Cloud Platform). Available functions are:
  - All facilities included in the Standard model
  - Virtual servers management
  - Using “Infrastructure as Code” tools - Terraform, AWS CloudFormation
  - Auto configuration
- **The Enterprise model** allows users to get access to public virtual cloud providers (AWS, Microsoft Azure, Google Cloud Platform) and private regions located on OpenStack and VMware platforms**.**
  - All functions included in the Professional Deployment model are available, and applicable to both public and private clouds.

1.3 PROVISIONING MODES

The Maestro application can be provisioned to the users in one of the following options:

- **SaaS** (Software as a Service). The software is hosted in cloud and is provided to the user by subscription. The user can connect his account to the application and get access to the functionality within the requested deployment model.
- **SaaS + Privat Agent**. The agent is installed in a user’s private region (OpenStack or VMware) to enable Maestro control over it. The agent ensures that Maestro, hosted in cloud, is connected to the customer’s private cloud. Due to agent settings, Maestro receives only the information, approved by the customer. If the customer also has infrastructures in public clouds, they can be added to Maestro and managed according to the requested deployment model.
- **On Premise**. The Maestro application is installed locally on an isolated instance in the customer’s enterprise data center.

Innovative in the On Premise model is the fact that the cloud-based application can be installed in a closed perimeter, without any other cloud services installed. The Maestro application is deployed in private clouds and is focused on protecting the management perimeter and preventing interactions with external cloud service providers. It is achieved due to the correct level of abstractions and a special application construction procedure and the ability to install the system in a closed perimeter without access.

1.4 TECHNOLOGY STACK AND INTEGRATIONS

Maestro uses the following integrations and third-party tools for its core elements:

M3 Component	Third-Party Component	Version
M3Server	Java (Amazon Corretto)	OpenJDK 64-Bit Server VM Corretto-11.0.17.8.1 (build 11.0.17+8-LTS)
	Python	3.9
	Monogo Java Driver	4.0.6
	MongoDB	5.0.9
	Terraform	0.14.7, 0.15.1
	RabbitMQ Java Client	5.9.0
M3Admin	Python	3.9
	AWS-Syndicate	3.8
m3UI part	Angular	14.5.2
	primeng	14.1.2
	rxjs	6.6.7
	seamless-immutable	7.1.2
M3 Native part	nativescript/angular	13.0.1
OnPrem	MinIO	minio/minio:RELEASE.2020-03-05T01-04-19Z
	Vault	vault:1.3.2
	Monogo Java Driver	4.0.6
	MongoDB	mongo:5.0.9
	AWS SDK	1.11.785, 2.13.19
	RabbitMQ Java Client	5.9.0
	RabbitMQ	rabbitmq:3-management
	Dagger	2.27
	Spring Boot	2.3.10.RELEASE
	Spring	5.2.14.RELEASE
	Syndicate plugin	1.8.0

1.5 MAESTRO SAAS PERMISSIONS

To work correctly, Maestro SaaS needs a specific set of permissions on the AWS account to where it is deployed.

There are three roles Maestro uses:

- eo_orchestrator – the main Maestro engine
- eo_instance – for Maestro server and Lambdas processing
- eo_api – for Modular (admin) API
- eo_lambda – for Lambdas processing

The **Maestro permissions** include those for the following services:

- AWS API Gateway
- AWS Lambda
- AWS SNS
- AWS S3
- AWS DynamoDB
- AWS Kinesis
- AWS Cognito
- AWS EC2
- AWS CloudWatch
- AWS CloudFront
- AWS Elastic Beanstalk
- AWS CloudFormation
- AWS Auto Scaling
- AWS STS
- AWS SSM
- AWS Athena
- Working with State Machines

For **Maestro Admin (Modular) Tool**, a trusted relationship document is needed in order to enable its access to the necessary tenants/accounts. For this, the home account should be specified as a trusted identity with the sts:AssumeRole permission.

In addition, two policies should be set up: one for creating resources and one - for removing. They affect the following services:

- AWS IAM
- AWS CloudWatch
- AWS SNS
- AWS EC2
- AWS STS
- AWS S3

The details of all the policies are given in Annex A - Maestro Permissions.

2 MAESTRO COMPONENTS

Maestro is a complex system, using a big set of components, facing specific needs and tasks of the application.

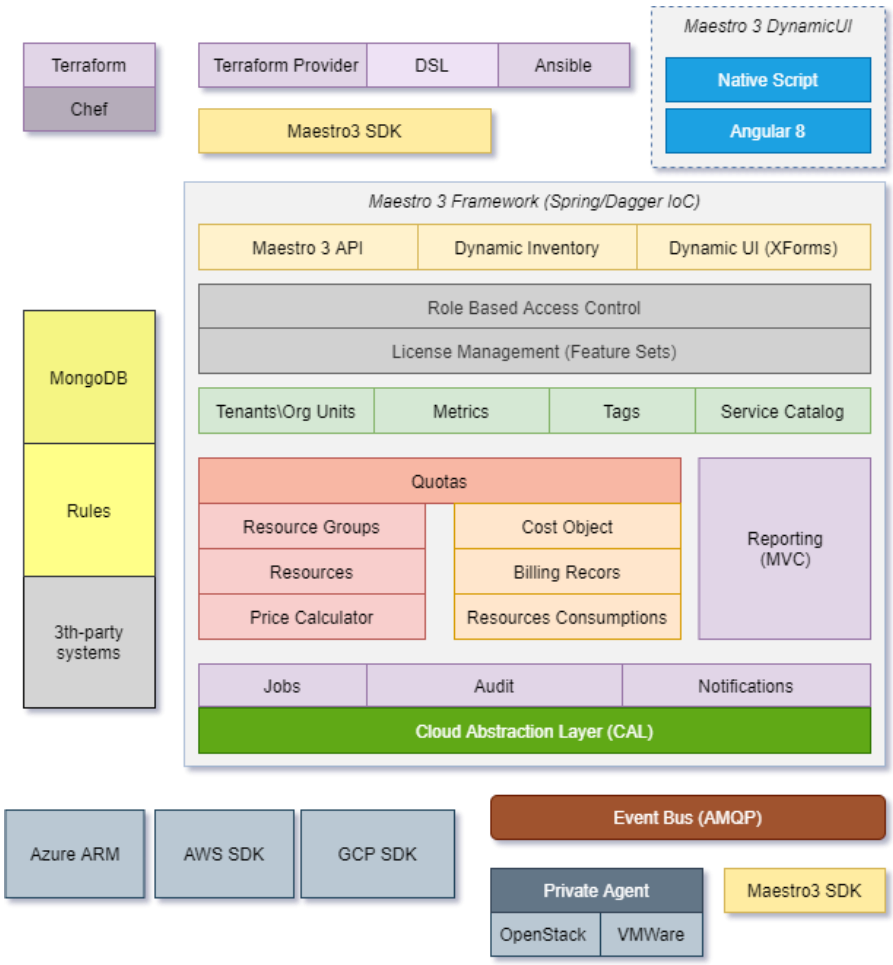


Figure 2 – Maestro components scheme

The main components of the solution are:

- Maestro Framework, allows to create applications, effective both in Lambda-based SaaS model and as on-premise solutions.
  The framework is based on Inversion of Control principle, and uses Spring and Dagger frameworks.
- Private agent – an application, allowing Maestro control over OpenStack and VMware-based private regions.
- On-Premise Installation – Maestro application, adapted for installing on customer's side.
- Cloud Abstraction Layer – the part of Maestro logic, responsible for integration with public cloud providers..
- Event-Driven Architecture – the set of components, responsible for tracking and interpreting the events in infrasturctures supported by Maestro.
- Maestro SDK – An SDK, enabling Maestro components integration.
- Dynamic UI – The effective dynamic User Interface, based on Angular 14 and Native Script.
- Maestro Databases – Maestro uses MongoDB and DynamoDB for its purposes.

- Support and DevOps tools – the set of tools, enabling effective solution support and automation, include Chef, Ansible, and Terraform.

3 MAESTRO FRAMEWORK

Maestro is a framework that allows to create applications, effective both in Lambda-based SaaS model and as on-premise solutions.

The framework is based on Inversion of Control principle and uses Spring and Dagger frameworks. Once Spring is typically used for solid applications, Dagger is convenient when the code is split into AWS Lambda functions.

Once the solution is Lambda-based, Spring framework is not effective from Lambda functions start time perspective. To enable higher effectiveness and productivity, Dagger framework is used to combine all elements of the system.

3.1 API-FIRST APPROACH

As part of the Maestro system, we follow the concept of API First.

Such a concept implies that at the beginning of any application development phase a certain API should be created. Then, on top of this API, we develop additional necessary modules.

In such a way the Maestro API was implemented, providing Maestro SDK as an additional module on top of it.

3.1.1 Maestro SDK

Initially the Maestro functionality is supported in the Maestro SDK, and only then it is implemented on the UI. Thus, SDK is the keystone for all Maestro functionality.

Maestro SDK is based on JSON RPC protocol - a remote procedure call protocol encoded in JSON. The protocol is similar to XML-RPC and defines a few data types and commands. JSON-RPC allows for notifications (data sent to the server that does not require a response) and for multiple calls to be sent to the server which may be answered out of order.

Based on this protocol, a simple and flexible working SDK architecture was created.

The user, working with Maestro SDK, contains a client. The client contains some managers. Managers contain some methods. The client is a ready-made entity that can do anything. Each of managers is responsible for a certain task, i.e. a certain manager for reporting, a manager for a private cloud, a manager for working with Terraform, etc..

In case a completely new functionality should be added in the system, a new manager is created. This manager is responsible for a specific new function. The methods required for performing this function are included to this manager as well.

The JSON RPC protocol is used in such a case, allowing to rapidly expand the functionality. Maestro SDK collects the necessary data and sends it to a server via HTTP or RabbitMQ protocols (both are supported in Maestro SDK). The server processes these requests and returns responses.

Security of data transfer is ensured by the AES encryption.

3.1.2 Ansible and Dynamic Inventory

Ansible is an automation system for provisioning and configuring infrastructure, allowing to install software on virtual machines. It can also manage large clusters of VMs on different cloud providers.

Maestro provides the possibility to generate a dynamically assembled list of virtual machines via API that is used by Ansible DI via self-service.

Ansible Dynamic inventory is a tool allowing setting up configuration for several instances. It requires a list of instances to be configured. Having the necessary list, Ansible can start working with it to configure the given instances.

Together with other Maestro features, the product allows solving task which can be covered by Ansible Tower.

Find more details in the Ansible section.

3.1.3 Dynamic UI

API First approach is engaged in creating Maestro dynamic UI. Dynamic forms concept was implemented based on such tools as Angular 14 on the web side and Native Scripts on the native side (IOS, Android) using the same backend API for web and mobile versions. More details about Dynamic UI and the tools used to support it, you can read in Dynamic UI section.

3.2 ACCESS CONTROL

The Maestro system is based on two principles of access control and management: Role-Based Access Control and License Management.

Role-Based Access Control approach is introduced by a permissions model based on assigning certain scopes of permissions to users on different levels: Customer, Tenant, Project, and User.

License Management is presented by the Feature Toggling functionality, allowing to manage users' access to the Maesptro3 application based on their subscriptions or purchased application editions.

3.2.1 Role-Based Access Control

In scope of Maestro application permissions Model, the classic RBAC model was extended. It now consists of three levels of entities: Role, Permission Group, and Action:

- **User Role** – a user attribute defining the user's general level of access to the Cloud. Influences user permissions. Based on the user's specified role on a tenant, the user is allowed to perform a scope of operations defined in one or several permission groups.
- **Permission Group** - a set of Maestro operations grouped by purpose and expected admission level of the users who will have access to the group. Each user has access to one or several permission groups depending on their role within the tenant.
- **Actions** - a scope of operations the user can perform in the cloud based on user's permissions.

Each User Role includes Permission Groups, each Permission Group includes Actions, each Action corresponds to some action that the user can perform in Maestro. Such a structure allows to customize the model based on customer's requirements and provide flexible access to users.

User Permissions Management

Permissions can be assigned on different levels, such as Customer, Tenant, Project, and User. Permissions are based on the project role of the user and defines what actions s/he can perform.

Currently, the permission concept is based on three basic notions - permission action, permission group, and permission mapping:

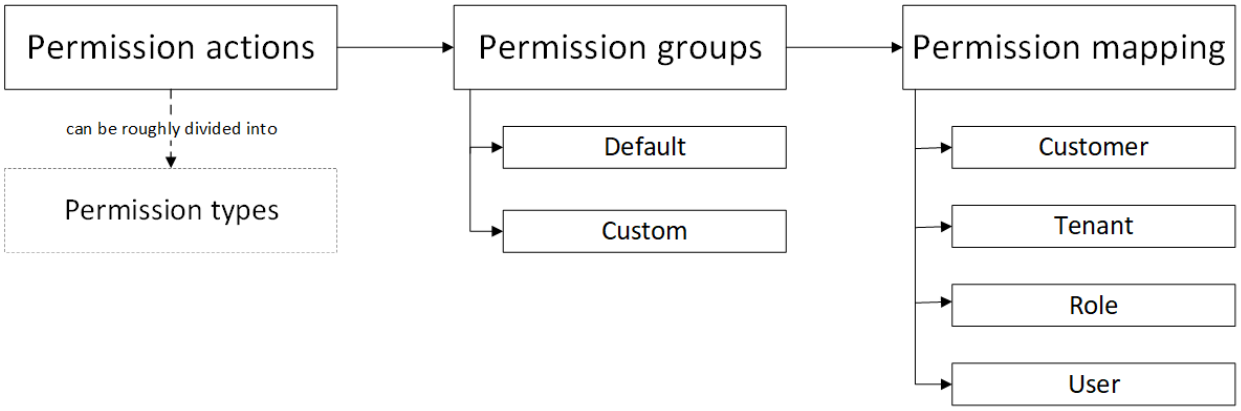


Figure 3 – User permissions management scheme

Permission action

Permission action is a basic notion that defines definite tasks that can be performed by the user.

For the sake of quicker and easier referencing, all permission actions can be roughly divided into three groups:

- wizard actions (e.g., CREATE_IMAGE_WIZARD)
- page action (e.g., IMAGES_PAGE)
- actual actions (e.g., CREATE_IMAGE)

Permission group

All permission actions are united into permission groups.

Permission groups can be default and custom.

Custom permission groups are created on request and can unite available permission actions in any combinations necessary for the customer.

Default permission groups are predefined:

- **Full_Access** permission group gives the full access (create, manage, read, and kill resources) to the full functionality of Maestro UI (SaaS).
- **Read_Only** permission group gives the read-only access (create, manage, read, and kill resources) to the full functionality of Maestro UI (SaaS).
- **Resources_Management** permission group allows creating, managing, reading, and killing resources related to virtual machines (except for billing).
- **Resources_Managment_ReadOnly** permission group gives the read-only access against creating, managing, reading, and killing resources related to reporting and billing functionality.

- **Billing** permission group allows creating, managing, reading, and killing resources.
- **Billing_ReadOnly** permission group gives the read-only access against creating, managing, reading, and killing resources.
- **Manage_Cloud** permission group gives the full access to security checks and permissions management.
- **Manage_Cloud_ReadOnly** permission group gives the read-only access to security checks and permissions management.
- **Basic** permission group gives the access to the basic Maestro UI functionality (support, using native consoles, managing themes, My permissions and Default settings, managing notifications).

Permission mapping

To determine definite actions that can be performed by a definite user, specific permission mappings are set up.

There are four permission mappings that are determined by their scope and the place in the mapping hierarchy:

- **customer** - permission actions available for all the users of a customer
- **tenant** - permission actions available for all the users for a tenant
- **role** - permission actions available for all the users with a specific role
- **user** - permission actions available for a definite user

The Hierarchy of permission mapping is this:

1. If a mapping exists for a definite user in a tenant, user mapping is used.
2. If a mapping exists for the role in the tenant and this role is not blocked in this tenant, role mapping is used.
3. If a mapping exists for the tenant, tenant mapping is used.
4. Otherwise, customer mapping is used.

3.2.2 License Management

When offering the Maestro application to a wide audience of users, the following aspects related to license payment should be considered:

- The Maestro solution is provided to users based on a set of features that users selected. Thus, it is necessary to ensure that the user who received access to the application, can use only those features which were selected and paid for.
- The Maestro application can be purchased in several editions, and depending upon the selected edition, it is required to be able to cut off the respective purchased/non-purchased functionality.
- It is also necessary to control access to the application functionality if a subscription (monthly or yearly) is provided to users. In case the regular payment is not performed, the Maestro application can be disabled from the client's side until the respective payment is made.

All these objectives are accomplished with the **Feature Toggling** functionality supported by Maestro. It provides the possibility to switch features on/off as well as ensure efficient control over subscriptions and monthly/ annual payments.

The Feature Toggling functionality will guarantee that the Maestro application will not be manipulated by users without our privity, and all copies of the applications will be under control by respective Maestro management and support teams.

3.3 ORGANIZATION STRUCTURE SUPPORT

Logical instruments to organize v-infrastructure close to the way it is done in your organization.

3.3.1 Organization Units

Maestro supports several organization units which allow for better organization and management of cloud resources.

The main organization unit for Maestro is a **tenant**. Tenant is a group of users independent of other groups but sharing the same cloud infrastructure and resources. A tenant may be a company or a department within a company or a customer, etc. All cloud providers have their equivalent for a tenant:

- accounts in AWS,
- subscriptions in Microsoft Azure,
- projects in Google Cloud Platform.

Tenant group is a logical unit of tenants created according to the customer's needs and internal structure if the company.

In terms of VMWare private agent (is described below), there is one more organizational unit, called **organization**. Organization is an entity within the VMWare vCloud Director that corresponds to a customer or its divisions that need separate resources in the cloud.

Maestro uses the multi-tenant approach in its billing system where different tenants are charged for the actually consumed resources. The maintenance costs are distributed between the private cloud tenants. The notion of tenants allows keeping full control of all the cloud resources and their costs.

It is recommended that every tenant has at least one separate account, subscription, and project (or different accounts for different environments) in different public clouds in order to make the resource management in this cloud provider more tractable.

Maestro supports these features as related to the organizational units of public cloud providers:

- several AWS organizations can be activated and are supported,
- different AWS organizations can be added to and supported by Maestro,
- Microsoft Azure subscriptions are supported,
- Azure Enterprise Agreement hierarchy model is supported.

3.3.2 Infrastructure Control

Once the infrastructure is built, you need to monitor and control it. To do it in an enhanced way, two types of KPI are available in Maestro:

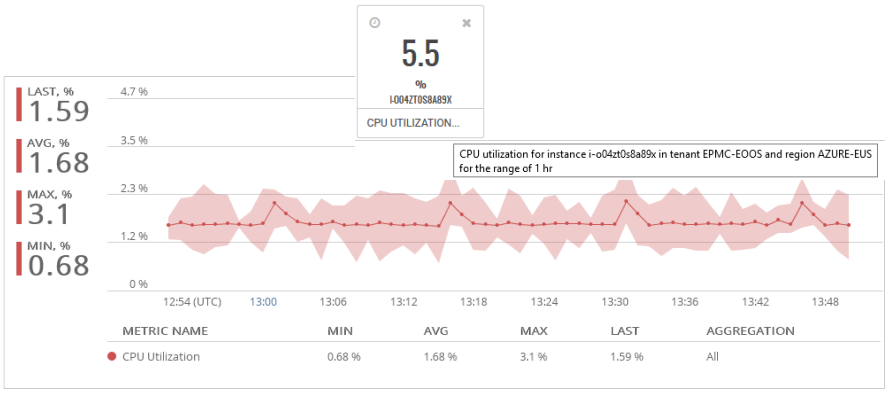
- Metrics showing performance to see if infrastructure is effective;
- Tags to structure work with infrastructure to make management, monitoring, and reporting more effective and convenient.

When talking about organizing infrastructure in a big enterprise, we know how to make parts of it clear, visible, and measurable.

Infrastructure Metrics

Maestro supports different kinds of metrics. There are three following groups:

1. By type:
   - Counters
   - Chargebacks
   - Instance-related
   - Custom
2. By dashboard:
   - Homepage
   - Management
   - Reporting
3. By layer:
   - Tenant
   - Region
   - Instance



Instance-Related Metrics

The table below show metrics for virtual machines which can be divided into different categories by cloud, OS and source.

Type	Unit	Cloud	OS	Source
CPU Utilization	Percent	AWS, Azure, Google	Linux, Windows	Built-in
Disk Read, Write	Bytes, Percent	AWS, Azure, Google	Linux, Windows	Built-in
Network Traffic	Bytes	AWS, Azure, Google	Linux, Windows	Built-in
Status Check	Bytes	AWS	Linux, Windows	Built-in
RAM Usage	Bytes, Percent	Azure	Linux, Windows	Extension
SWAP Usage	Bytes, Percent	Azure	Linux	Extension

Built-in (host-level) metrics are host computer metrics. One of the examples is the CPU utilization metric.

There are also metrics for guest VMs (**guest-level** or **extended** metrics). Examples of extended metrics include Memory and SWAP usage.

Extended Metrics on Azure

To enable extended metrics, you can take one of the flows:

- enable the **Extended Metrics** extension on the **Run** wizard marking the respective checkbox;

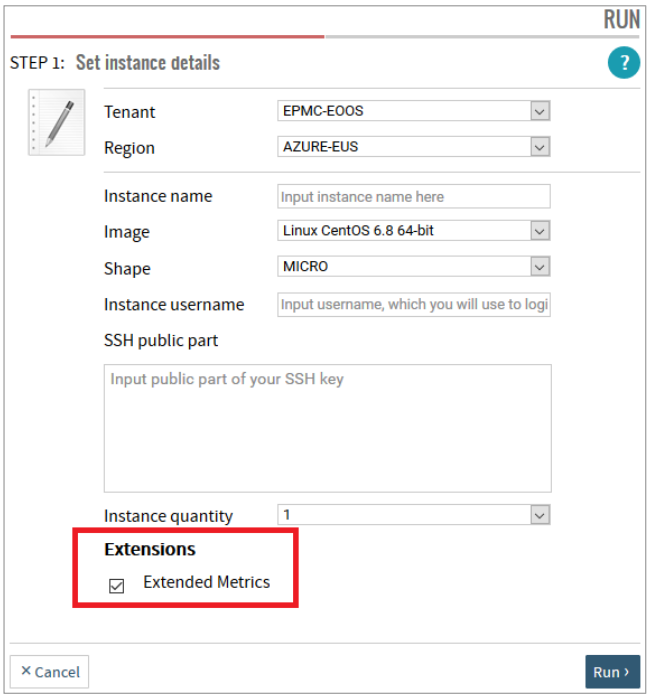


Figure 4 – Enabling extensions

- use the **Metrics extension** content-view item to install or remove the extension.

Figure 5 – Installing or removing Metrics extension

You can also enable extended metrics on Azure Portal through the **Diagnostic Settings** section of VM details.
View them under **Monitoring/Metrics** by selecting **Guest (Classic)** in **Metric Namespace**.

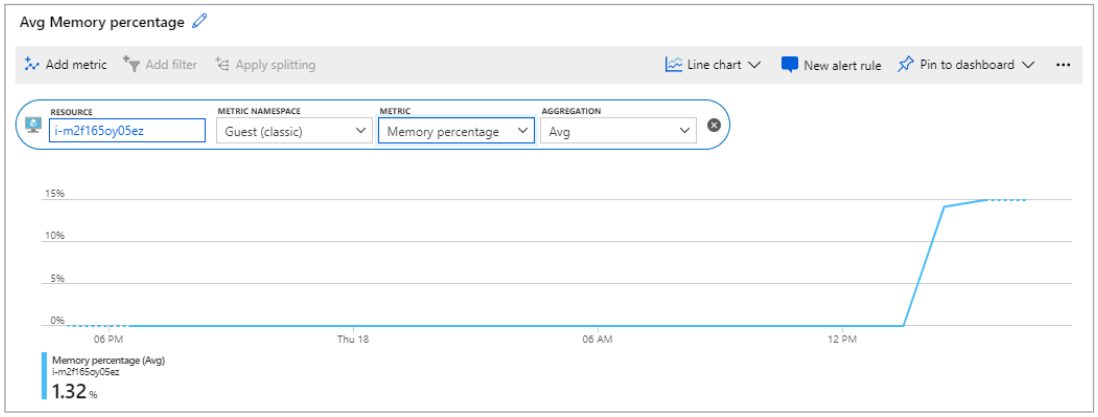


Figure 6 – Memory metrics

Enabling guest-level monitoring installs the Azure diagnostics agent on the VM. By default, a basic set of extended metrics are added (memory and SWAP usage). The process is the same for Windows and Linux VMs.

Please note that extension can be only installed on Linux and Windows instances in running state. Extension should be configured for usage same storage account, that our application. Possible extension states:

- available to install
- installed
- not installed
- not supported
- installed, but with unsupported storage account configuration

In the last case, re-install the extension to interact with the Maestro application. The "Not supported" state is possible on OS other than Linux and Windows, or on very old versions (currently, not handled).

Visualization

The application provides graph viewing and adding metrics only for metrics supported by the selected virtual machine. This affects the availability of graphs in the list of virtual machines on management page and the presence of metrics on the corresponding wizard.

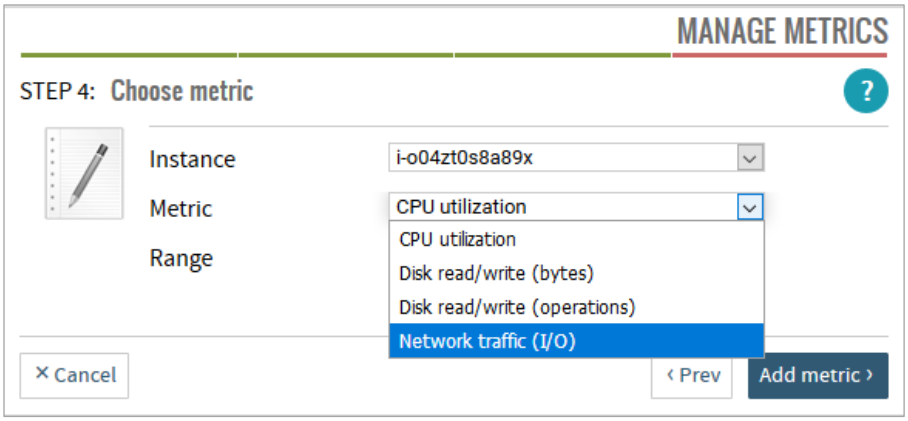


Figure 7 – Adding metrics

Storage Account

Guest-level monitoring requires a storage account to store information about extended metrics. Azure offers several types of storage services. By default, metrics are stored in Table Storage Service, which allows to use CosmosDB queries for retrieving the data. The Maestro application creates one storage account for each region in each subscription. Information from virtual machines can only be recorded in the storage account of the corresponding region.

You can obtain access to a storage account generating an authorization key (2 per account, permanent access until recalled), or via shared access signatures (unlimited amount, limited access duration). See an example of an endpoint below:

```
http://<storage_account>.table.core.windows.net.
```

A storage account is billed based on its usage – the amount of stored data and queries.

Resource Tags

Tag is a type of meta information that can define access control rules and describe access requests. Based on that, it can be applied to any entity, e.g. a resource to work for all Cloud providers.

Adding a tag to an entity, you can also request all resources grouped by it. When sending such a request, you can get the whole variety of entities available from Cloud provider via one API. Thus, tags are an instrument for analysis, allowing to work with Cloud, and provide resource and cost management in an effective way.

Maestro uses tags in terms of a unified approach to:

- discover resources
- group resources into logical groups
- control billing
- control access

You can find a panel with filters on the Management page. The availability of filters may vary depending on the selected cloud.

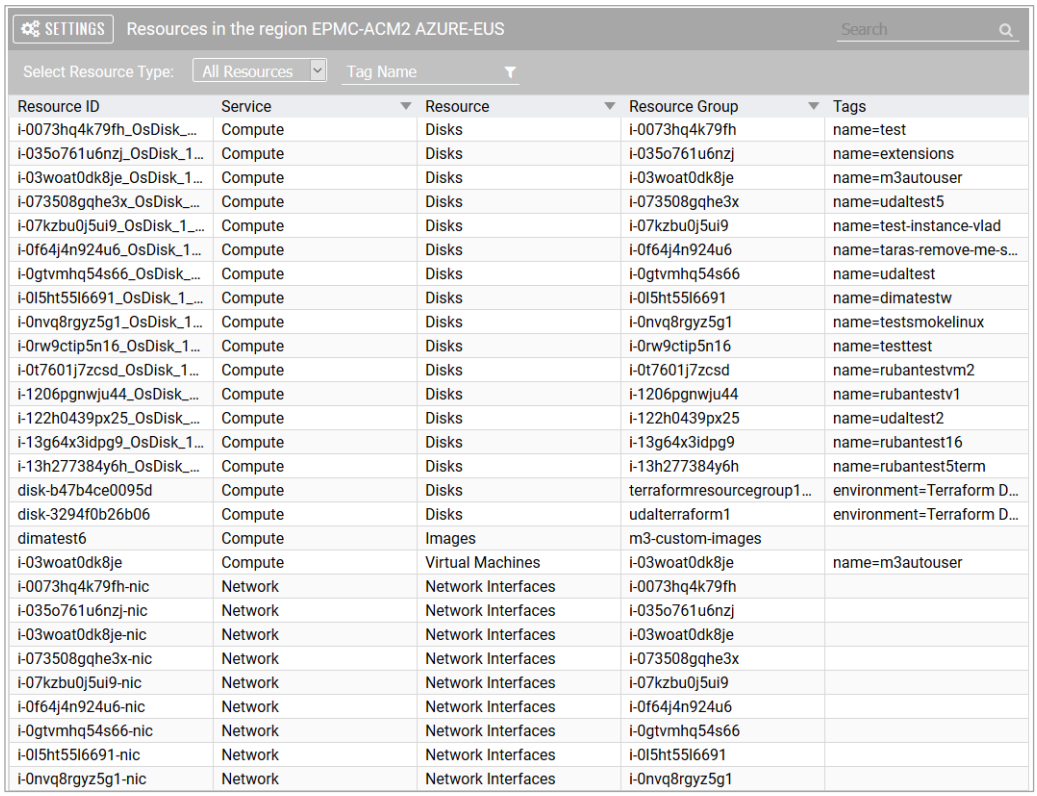
Filters can be nested, for example, filter by Resource Type includes a filter by Tag Name. They can be in form of select items or text inputs.

Currently, filters on Google and OpenStack are in progress and not available.

The list below shows filters hierarchy by Clouds:

- AWS:
  - Filter by Resource Type
    - Instances (default)
    - All Resources (includes all AWS resources; columns: ResourceID (*short*), Service (*e.g. EC2, CloudWatch*), Resource (*e.g. Instance, Rule*), Tags (*as key=value pairs*))
      - Filter by Tag Name (exact match, case sensitive)
- Azure
  - Filter by Resource Type
    - Instances (default)
    - All Resources (includes all Azure resources; columns: ResourceID (*short*), Service (*e.g. Compute*), Resource (*e.g. Disk, Image*), Tags (*as key=value pairs*))
      - Filter by Tag Name (exact match, case sensitive)
- Google
  - Filter by Resource Type
    - Instances (default)
    - All Resources (includes only Instances, Images, Volumes; columns: ResourceID (*short*), Service (*e.g. Compute*), Resource (*e.g. Disk, Image*), Tags (*as key=value pairs*))
      - Filter by Tag Name (exact match, case **in**sensitive)
- OpenStack
  - Filter by Resource Type
    - Instances (default)
    - All Resources (includes only Instances, Images, Volumes; columns: ResourceID (*short*), Service (*e.g. Compute*), Resource (*e.g. Disk, Image*), Tags (*as key=value pairs*))
      - Filter by Tag Name (exact match, case insensitive)

See an example of filtering resources by tag below:



Resource ID	Service	Resource	Resource Group	Tags
i-0073hq4k79fh_OsDisk_...	Compute	Disks	i-0073hq4k79fh	name=test
i-035o761u6nzj_OsDisk_1...	Compute	Disks	i-035o761u6nzj	name=extensions
i-03woat0dk8je_OsDisk_1...	Compute	Disks	i-03woat0dk8je	name=m3autouser
i-073508gqhe3x_OsDisk_...	Compute	Disks	i-073508gqhe3x	name=udaltest5
i-07kzbu0j5ui9_OsDisk_1_...	Compute	Disks	i-07kzbu0j5ui9	name=test-instance-vlad
i-0f64j4n924u6_OsDisk_1...	Compute	Disks	i-0f64j4n924u6	name=taras-remove-me-s...
i-0gtvmhq54s66_OsDisk_...	Compute	Disks	i-0gtvmhq54s66	name=udaltest
i-0l5ht55l6691_OsDisk_1_...	Compute	Disks	i-0l5ht55l6691	name=dimatestw
i-0nvq8rgyz5g1_OsDisk_1...	Compute	Disks	i-0nvq8rgyz5g1	name=testsmokelinux
i-0rw9ctip5n16_OsDisk_1...	Compute	Disks	i-0rw9ctip5n16	name=testtest
i-0t7601j7zcsd_OsDisk_1...	Compute	Disks	i-0t7601j7zcsd	name=rubantestvm2
i-1206pgnwju44_OsDisk_...	Compute	Disks	i-1206pgnwju44	name=rubantestv1
i-122h0439px25_OsDisk_...	Compute	Disks	i-122h0439px25	name=udaltest2
i-13g64x3idpg9_OsDisk_1...	Compute	Disks	i-13g64x3idpg9	name=rubantest16
i-13h277384y6h_OsDisk_...	Compute	Disks	i-13h277384y6h	name=rubantest5term
disk-b47b4ce0095d	Compute	Disks	terraformresourcegroup1...	environment=Terraform D...
disk-3294f0b26b06	Compute	Disks	udalterraform1	environment=Terraform D...
dimatest6	Compute	Images	m3-custom-images	
i-03woat0dk8je	Compute	Virtual Machines	i-03woat0dk8je	name=m3autouser
i-0073hq4k79fh-nic	Network	Network Interfaces	i-0073hq4k79fh	
i-035o761u6nzj-nic	Network	Network Interfaces	i-035o761u6nzj	
i-03woat0dk8je-nic	Network	Network Interfaces	i-03woat0dk8je	
i-073508gqhe3x-nic	Network	Network Interfaces	i-073508gqhe3x	
i-07kzbu0j5ui9-nic	Network	Network Interfaces	i-07kzbu0j5ui9	
i-0f64j4n924u6-nic	Network	Network Interfaces	i-0f64j4n924u6	
i-0gtvmhq54s66-nic	Network	Network Interfaces	i-0gtvmhq54s66	
i-0l5ht55l6691-nic	Network	Network Interfaces	i-0l5ht55l6691	
i-0nvq8rgyz5g1-nic	Network	Network Interfaces	i-0nvq8rgyz5g1	

Figure 8 – Filtering resources by tag

You can also use tags for billing reporting. They are different for different Cloud providers:

- AWS - cost allocation tags
- Azure – tags
- Google – not supported.

*Please note that these words are reserved by Azure and cannot be used as a tag prefix or a tag key – **microsoft**, **azure**, **windows**:*

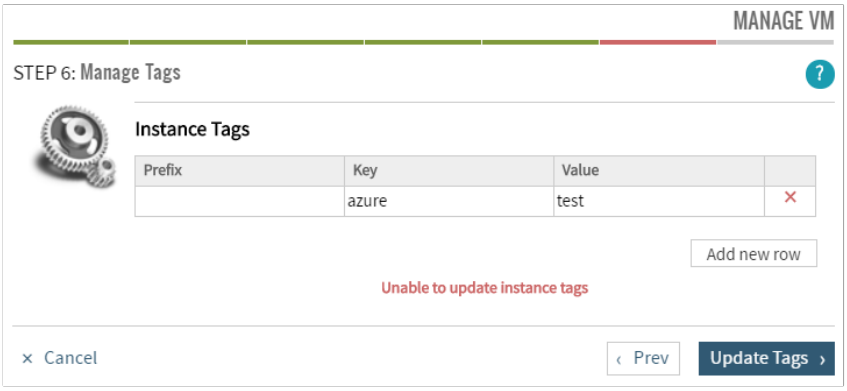


Figure 9 – Manage tags window

3.3.3 Service Catalog

Maestro provides access to the **Service Catalog** for its internal customers and potentially these services can be present on the Marketplace for external ones. Such services as Cloud Management tools, Cloud hosting services and platform services are available for the customers.

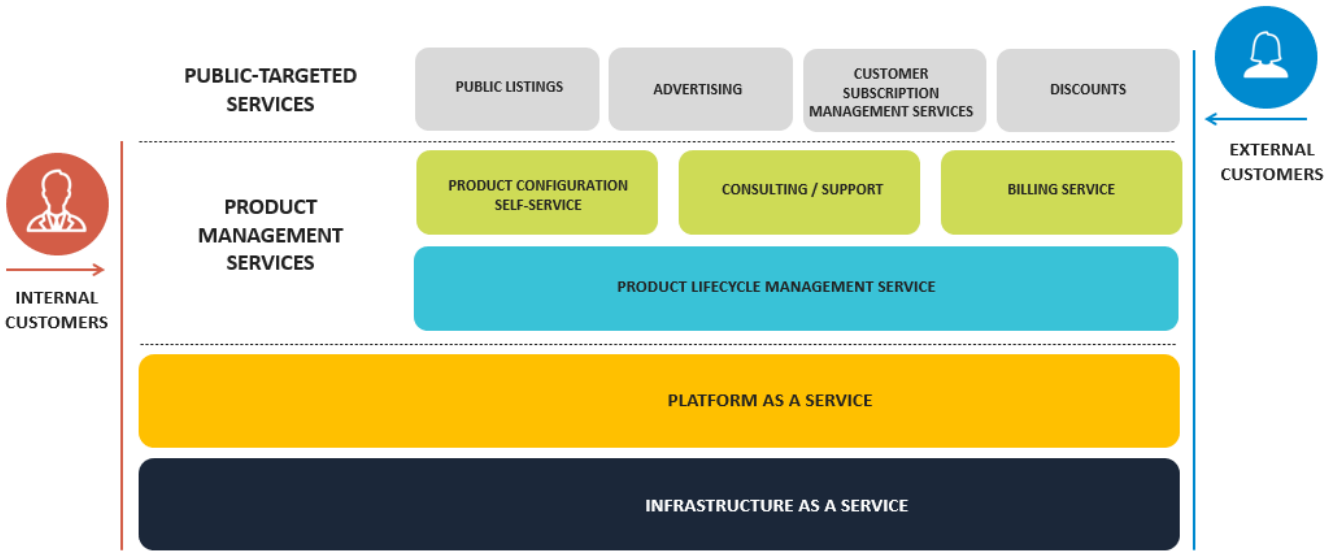


Figure 10 – Marketplace components overview

A service has its lifecycle that includes service activation, update, and deactivation. Each stage of its lifecycle is monitored and analyzed by the system. To enables access to the Service Catalog and control the product's lifecycle Maestro has enough capabilities to ensure this process form both business and technical points of view.

The main components provided by the system that are required for the reliable services product management include:

- **Product configuration in self-service**, that allows users to adjust the product according to the project needs.
- **Support and consulting service** that provides assistance to the customers who face with difficulties and problems.
- **Billing service**, that allows to bill all the resources consumed.

In current version, Service Catalog is available for Maestro internal customers. The possibility to put supported services to public marketplaces, in order to make the product easily recognizable and available for external customers, is under development.

These services include public **listening, advertising, customer subscription management services and discounts.**

Maestro is built on Infrastructure as Code concept. Service catalog uses **Terraform** as a main tool for service creation and deployment as well as **Catalog** to provide access to the Terraform service. This approach allows to deploy dynamic infrastructures. Terraform is a unified tool to create and configure infrastructure for different cloud providers. It allows to plan the action that can be performed for the infrastructure that we are going to deploy and predict infrastructure costs. Terraform provides full product lifecycle management.

Using Terraform allows Maestro to be Cloud Antagonistic as users can create templates for different cloud providers. Maestro includes Terraform Provider for M3 which uses M3 SDK, which allows to create cross-cloud templates and services.

More details about Terraform integration with Maestro you can find in this section.

Maestro service adding flow includes the following steps:

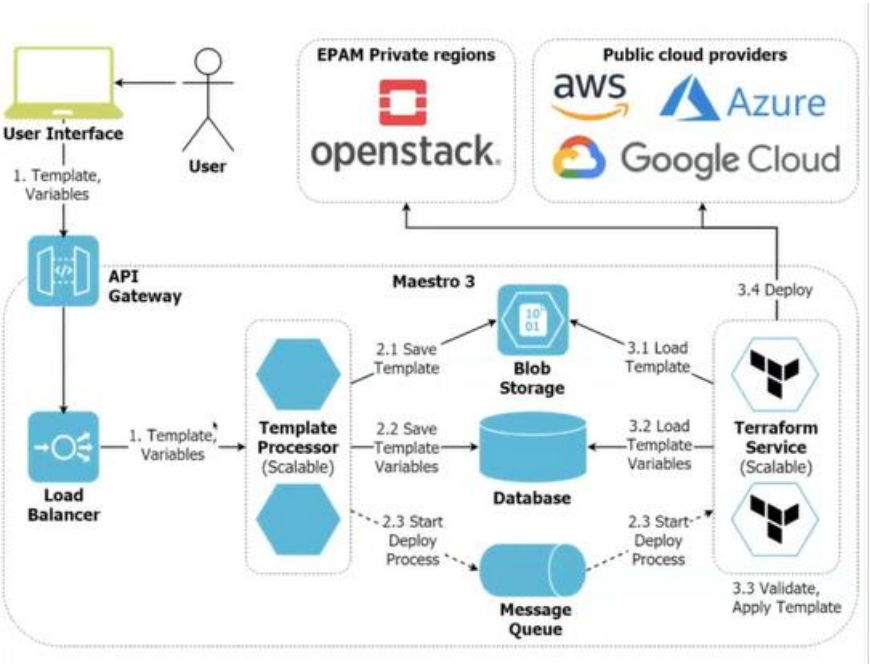


Figure 11 – Adding service flow

1. **User** defines product template and variables trough user interface
2. **Template processor:**
   - Saves the template to the blob storage
   - Saves variables in a database
   - Starts a deploy process to Terraform Service through a message queue
3. **Terraform Service:**
   - Loads template from blob storage
   - Loads variables from database
   - Validates and applies the template
   - Deploys service to public or private cloud

The Catalog page on the User portal looks as follows:

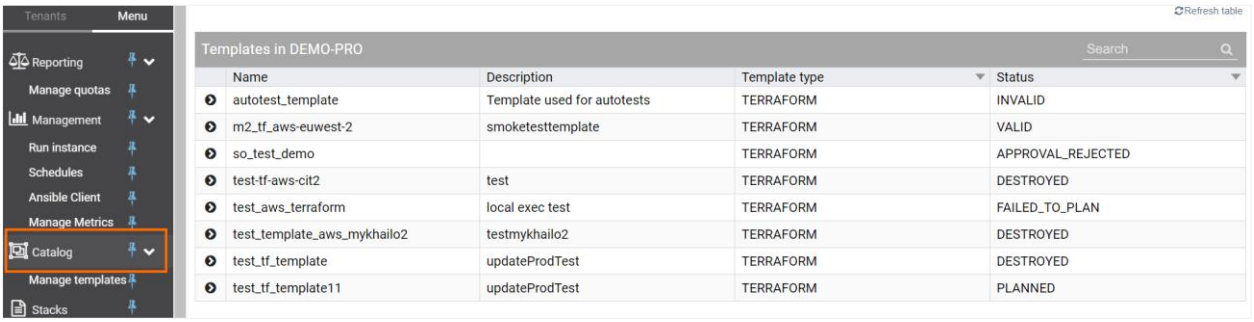


Figure 12 – Catalog page

3.4 BILLING AND QUOTAS

Maestro provides a unified billing service that enables collecting private and public billing data for tenants with different virtual infrastructures and displaying it to the user in real-time mode.

All Cloud-related bills are assigned to the tenant to which the respective resources or services belong. You cannot reassign any bills to another tenant.

3.4.1 Quotas

Every tenant is subject to quotas that specify how many resources can be used by the tenant within a definite period of time. The aim of these quotas is to control the spending of the tenant's financial assets and to ensure that its cloud resources are used with the optimal efficiency.

Maestro supports two types of quotas:

- **daily resource quotas** limiting the resources that can be created on the tenant in one day.
- **monthly project quotas** specifying the tenant's monthly cost limit for the cloud usage.

Daily resource quotas

Daily resource quotas specify the amount of resources that can be created on the project within 24 hours. These quotas can be set automatically and result from cloud infrastructure capacities.

Monthly tenant quotas

Monthly project quotas specify the sum of money the tenant can spend on its cloud resources per month.

There are three types of monthly tenant quotas that differ in their scope:

- **all** is a quota applied cumulatively to all regions in which the project is activated.
- **cloud** is a quota applied to each of the regions of a certain cloud provider in which the project is activated (separate for EPAM, AZURE, AWS, and GOOGLE).
- **region by name** is a quota for a specific region.

Besides setting the maximal allowed monthly costs for the tenant resources, monthly tenant quotas specify:

- **action plan** that is what happens after the quota level is reached (possible options are *Stop instances when 100% quota depleted*, *Request approval for new instances after quota is 100% depleted*, and *Deny requesting new instances when 100% quota is depleted*),
- **notification plan** that is when you will get the email notification about your quota usage (if you set notification plan to 90, this means that when 90% of your quota is spent, primary and secondary contacts of the tenant will get an email saying that 90% percent of the tenant quota is spent). The notification for 100% of quota spent is sent in all cases regardless of any notification plan settings.

Monthly tenant quotas be changed by the tenant members with the necessary permissions (UPDATE_QUOTA) in the **Manage Quotas** wizard.

Here is a diagram that illustrates how the quotas are applied:

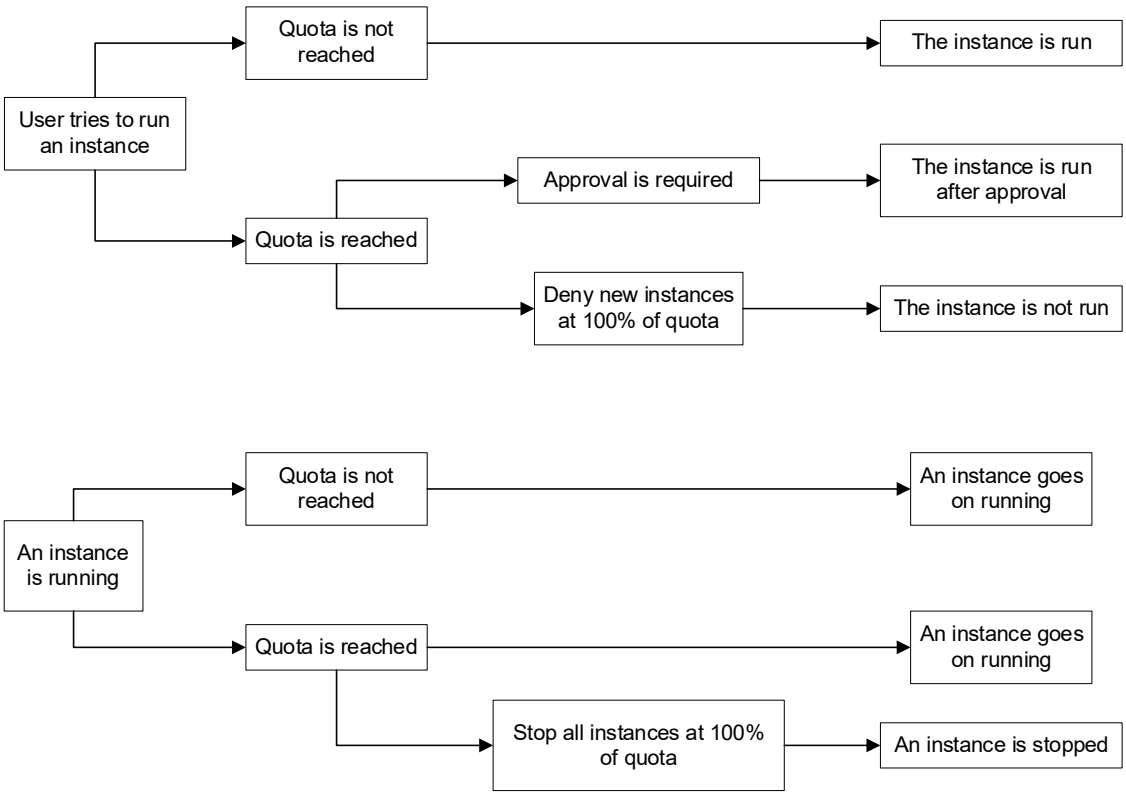


Figure 13 – Application of monthly tenant quotas

3.4.2 Billing

Billing system for private regions is based on the **pay-as-you-go** principle that allows strictly controlling infrastructure configurations and expenses and releases constraints implied by prepaid models.

The basic rules of the **pay-as-you-go** principle are as follows:

- Resources can be billed on per-second (OpenStack) or hourly (VMWare) basis. The price for a second/hour is derived from the pre-established monthly price and assumes that there are 730.5 hours per month on average.
- Billing is different for running and stopped VMs, as the stopped VMs consume less resources.

The price of the solution depends on the selected cloud, instance configurations, and usage patterns (how many hours per day/month your VMs are up and running).

Cost object is the description of the service that is utilized by a cloud user and is subject to the billing. Cost objects can be different – for example, instance usage, storage usage, requests, etc. Cost object is one of the core concepts in the Maestro billing.

In Maestro, there are two types of billing – **private** and **public**.

- **Private billing** is realized through the private agent. Private agent generates events that are then billed according to their timelines.

  Timeline is an entity that describes the change of states for a certain resource during a period of time. For example, when the instance is started, the private agent begins a timeline; when the instance is stopped, the private agent closes the current timeline but is ready to start a new one, once the instance is started again; and when the instance is terminated, the private agent closes the timeline, and Maestro stops billing this instance.

Billing records are created hourly (at the break of each hour) based on the resource's timeline.

- **Public billing** is based on the billing records produced by the cloud provider. This is CSP invoice for Microsoft Azure; SQL queries to Amazon Athena for AWS, and data provided by the BigQuery service for Google Cloud Platform (also SQL-query-based).

  Maestro transforms these billing data into daily tenant records. Maetro3 doesn't store the actual billing data from public clouds but only uses them for creating its own billing records: Billing data from AWS and Google must be specifically queried, and Azure data is provided as is, but Maestro refers to them only when cloud users initiate the or2report action (or at the end of a billable period).

  Daily records created by Maestro are stored in its databases (so that a user will receive them as soon as needed) and then aggregated into weekly and monthly ones.

3.4.3 Price Calculator

Price calculator is a reference entity that includes the pricing schema for a definite configuration of a cloud resource. Price calculators are based on pricing policies and are used for creating hourly billing records for private regions. These is only one active pricing policy for every private region at a certain period of time.

There are different price calculators which allow for an extremely flexible billing mechanism:

- for instances,
- for volumes,
- for hardware services,
- for hardware devices
- for machine images,
- for checkpoints,
- service price calculator,
- mobile price calculator,

At the break of each hour, Maestro takes all the timelines open within an hour, analyzes the parameters of the billed cloud recourses (e.g., CPU and RAM for instances), and calculates the prices of these resources according to the active pricing policy by means of the relevant price calculator:

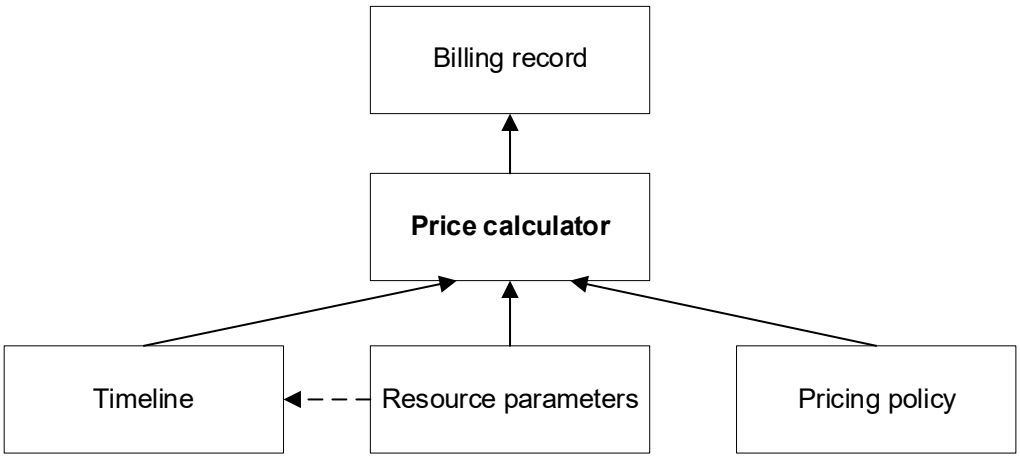


Figure 14 – Price calculating algorithm

3.5 USER COMMUNICATION

Maestro supports different channels of user communication in order to make it convenient and effective:

- Support requests,
- Emails and notifications (also available at the Notification page in the web version of Maestro and its mobile application),
- Push notifications from Maestro Mobile.

All these communication channels are equally supported by Maestro and its support teams but differ in the convenience for the user, user reaction time, and response time:

- up to 1 week for support requests,
- 1-2 days for simple letters,
- several hours for letters with approval,
- several minutes for push notifications in the mobile application.

3.5.1 Notifications

Notifications are the main way of communication between EPAM Cloud Orchestrator and its users, who are already got accustomed to receiving and reviewing notifications from their emails.

EPAM Orchestrator processes all information and delivers the most important messages (security, billing, costs, etc.) to the users, as well as carries out messages going via the communication channel between client and project, with the help of Maestro.

However, Maestro offers users an even more useful set of features ensuring that obtaining, filtering, and reviewing letters will become more efficient.

This diagram illustrates the Maestro notifications system:

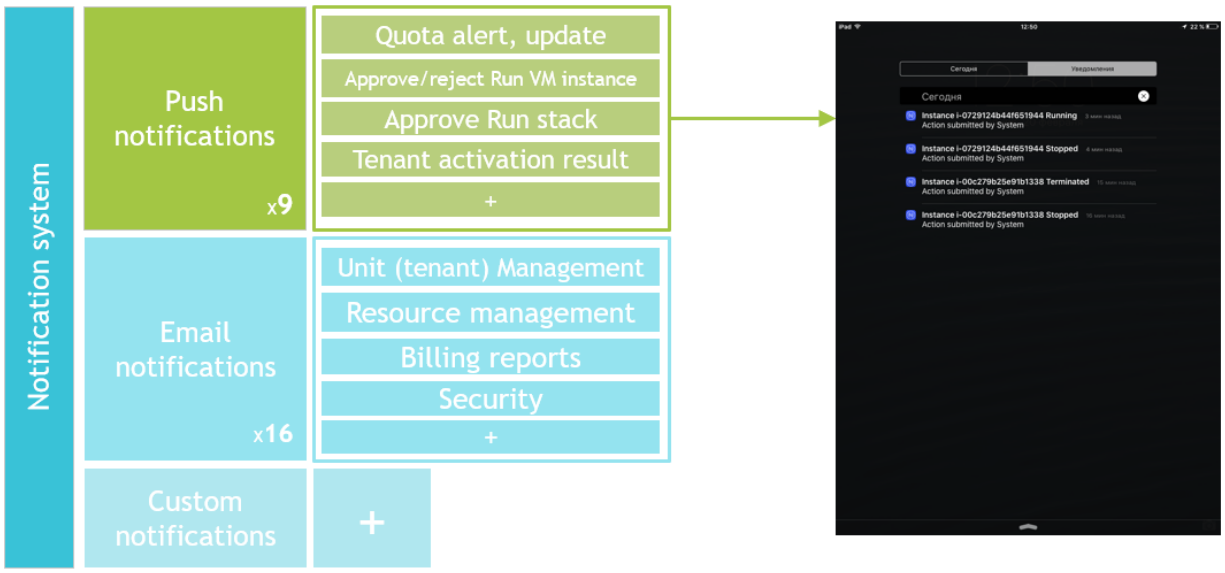


Figure 15 – Maestro notifications system

Notifications system in Maestro is based on the Spring MVC framework which means that actual data is separated from its representation. This allows changing the notifications appearance and formatting without changing the data model and employ free market templates for more compelling data representation. A

specifically developed renderer acts as an intermediary actor between the notification content and its representation: it takes the actual data and inserts it into the necessary places of the notification template.

This diagram illustrates how notifications are formed:

Figure 16 – Notification forming algorithm

Maestro stores all the notifications which have ever been sent through it in a model form that allows reaching two nearly exclusive goals – to save the storage capacities and to give the users quick and easy access to all their notifications.

Maestro provides the possibility to store the received notifications in one place.

In the **Notifications** page of the Maestro application, you will be able to review all notifications sent to them regardless of the period when they were received. Different filters provided on the interface will allow you to sort notifications to be displayed based on their content, priority, and time of receipt.

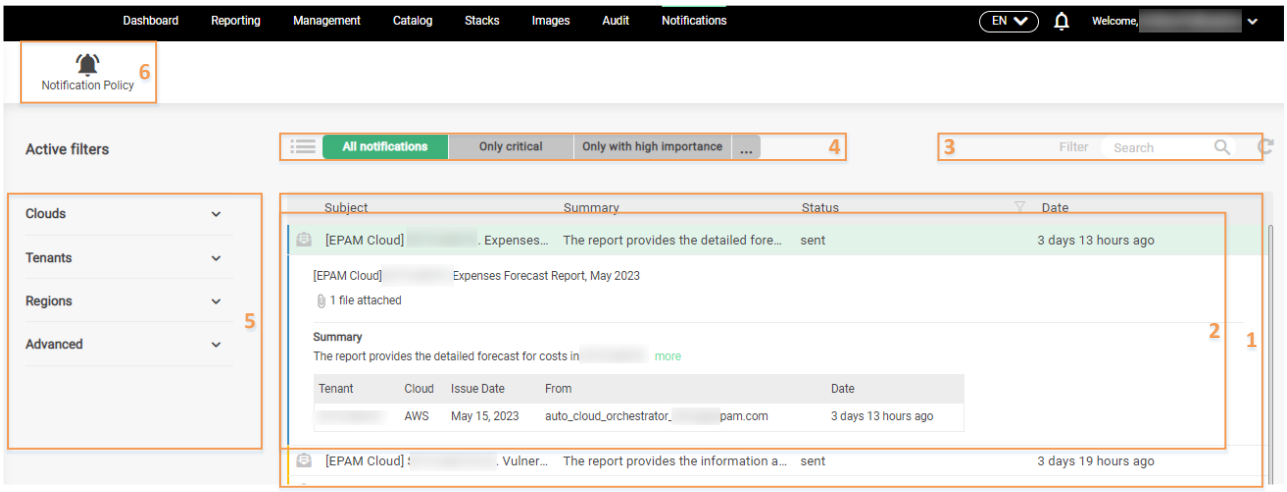


Figure 17 – Notifications page

In the Notifications page, the user can also find and review the notifications which were not sent to him/her after they disabled notifications or configured the default notification subscriptions.

Push notifications functionality is one of the outstanding features provided by Maestro Mobile Application. They allow you to immediately recognize and instantly respond to the most important events in the infrastructure and always be up to date with your infrastructure status.

Push notifications cover infrastructure events and approval requests and are grouped by these types:

- **Instance State Changed.** Sent to instance owners when the instance status changes (stop, start, reboot, terminate).
- **Login Approved.** Sent after the login attempt was approved by the responsible person.
- **Approved / Rejected Request.** Sent if the specific user action was approved or rejected.

3.5.2 Event Audit

All state-changing and other instance-related events occurring within Maestro are registered by the Audit system and saved to the event base. Audit system is vital for both the correct functioning of Maestro and the proper usage of cloud resources.

All the audit events are reported to the user in the form of email and push notifications described above.

Audit system is also the core element of the billing system because timelines according to which cloud resources are billed are formed on the basis of the audit events.

Maestro users can review the audit data in the Audit page of the Maestro web version.

3.5.3 Jobs

Job is an action performed by the system on behalf of the user or without their direct instructions. The most frequently performed jobs are:

- starting, stopping, and terminating instances according to the schedules,
- sending reports related to infrastructure state or billing,
- performing health and vulnerability checks and reporting their results to the user automatically or on request,
- monitoring and deleting system files, etc.

Jobs can be reviewed and monitored on the JMX page, a system management page that provides the information about different system elements and their functioning. In the JMX page, the user can see this information:

- jobs, their statuses and parameters, execution periods and schedules,
- application nodes, i.e. available servers together with their addresses, configurations, and certificates, server statistics, server launching parameters and utilization period, tasks assigned to these servers,
- server billing data.

JMX page is available only for users with definite project roles.

4 PRIVATE AGENT

Maestro makes the necessary provisions for working with both private regions and public clouds. These provisions are based on the notion of a private agent.

Private agent is a Java application that processes Maestro SDK requests and uses these requests to manage clouds (only those supported by a private agent). SDK requests are sent according to the AMQP protocol and encrypted using the AES (Advances Encryption Standard) algorithm.

Within the Maestro Enterprise solution, utilization and management of private clouds can be performed in two ways:

- via the OpenStack private agent used for managing OpenStack regions,
- via the private agent based on the VMWare technologies and installed on customer premises (VMWare private agent).

Maestro uses the same mechanism for interacting with both private agents. This mechanism involves RabbitMQ, a message broker installed on the Maestro side and used for connecting Maestro with private agents via the standard HTTP protocol. RabbitMQ is open to the world, while a private agent is located on a private network with Internet access connected to RabbitMQ. An HTTP / SSL connection is configured on the instance where RabbitMQ is installed, and then a connection is established between the private agent and Rabbit. These two systems interact by using standard message-broking procedure.

This diagram shows how Maestro communicates with private agents:

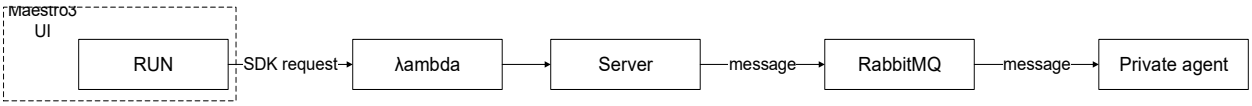


Figure 18 – Maestro-to-Private agent communication diagram

Private Agent is available within Enterprise deployment model of Maestro.

4.1 VMWARE PRIVATE AGENT

VMWare private agent is provided to customers with top security requirements so that they can be sure that no credentials are sent to or stored with third parties. MongoDB is used as a repository where VMWare private agent stores data required for the interaction with cloud. MongoDB is installed on the same instance on which the VMWare private agent is deployed and can only be accessed from this instance.

VMWare private agent manages the cloud by means of vCloud Director.

4.1.1 vCloudDirector Data Model

vCloud Director is a high-level API that directly allows setting up and managing clouds namely working with networks, providing disk spaces, managing instances, etc.

Besides this, vCloud Director supports:

- Instance Audit functionality where any instance action and status changes are reflected as audit events.
- White Listing option that allows assigning a public address by which Maestro will access to instances in a private network (in cases when some resources are made public).

vCloud Director is a complex entity that includes organizations and virtual data centers serving as containers for other resources.

Here is the structural diagram of vCloudDirector:

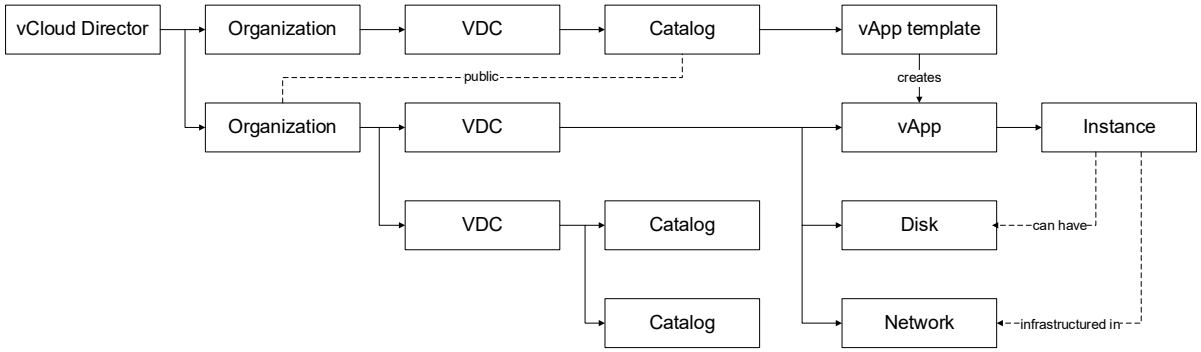


Figure 19 – vCloudDirector structural diagram

Organizations

Organization is an entity within the vCloud Director that corresponds to a customer or its divisions that need separate resources in the cloud. Every organization has its own virtual data centers (VDC), catalogs, vApp templates, and VMs.

Every organization must have at least one VDC. Without VDC, organization is an abstract entity with which nothing could be done besides logging in.

Currently, VMWare private agent supports only system access to vCloud Director (= credentials are used to access all the organizations). In future releases, separate access for different Organizations will be implemented (with separate credentials).

Organizations are used for managing resources in the sense that maximal time allowed for using VMs is assigned for an organization.

Virtual Data Center (VDC)

Virtual Data Center (VDC) is a virtual space where all the infrastructures are set up. In plain terms, VDCs serve as a container for everything.

VDCs contain catalogs (both public and private), vApp templates, disks, and networks:

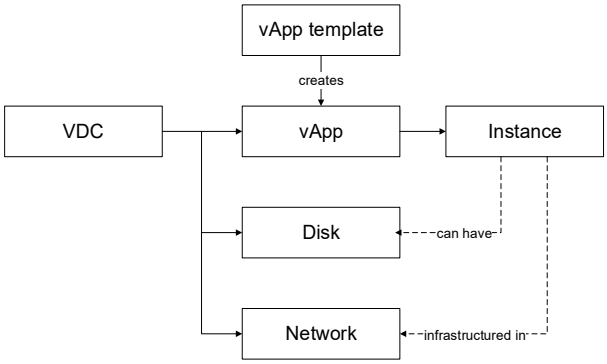


Figure 20 – VDC structural diagram

Each VDC needs some CPU, some RAM, and some amount of the disk space. The amount of VMs that can be run in a VDC is limited and is limited by the VDC capacities.

Catalogs

Catalog is a domain of image names. Catalogs are owned by organizations and are used for storing or sharing resources.

Catalogs can be private and public:

- Private catalogs are accessible only within the organization.
- Public catalogs are shared by several different organizations or all the organizations within this vCloudDirector.

Catalogs include vApp templates used for running VMs.

Every resource included in the catalog must have a unique name.

vApp Templates and vApp Applications

vApp template is a native VMWare infrastructure template that is used for running virtual machines. vApp templates ensure that VMs are consistently configured across an entire organization.

vApp templates include an operating system, applications, and data (similar to Terraform or CloudFormation templates).

No instance can be run without a vApp template.

vApp templates are not directly running virtual machines. Virtual machines are run by vApp applications created from vApp templates:

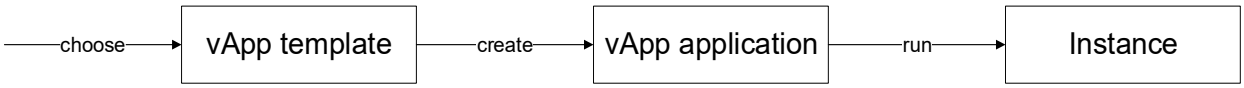


Figure 21 – vApp template-to-instance relation diagram

Currently, **one vApp template** can run only **one virtual machine**.

In future, the functionality of running more instances by one vApp template will be implemented.

4.1.2 Working with VMWare Private Agent

Current implementation of VMWare private agent supports only the most basic operations with virtual machines, but the available list will be expanded in future releases.

Available Instance Capacities

By default, Maestro users can get project resources of certain predefined capacities defined by shapes. Shape is determined by the number of vCPUs and the RAM memory volume.

VMWare does not have shapes of its own, so Maestro shapes are used for the sake of reference.

When you run an instance, some default capacities are suggested to you in the selected vApp template based on the Maestro shapes. You can choose this default suggestion or modify it to your necessary parameters of CPU and RAM.

Here is a reference table with shapes supported by Maestro:

Cloud Shape	CPU	RAM, GB
MICRO	1	0.5
MINI	1	1
SMALL	1	2
MEDIUM	2	4
LARGE	2	8
XL	4	7.5
2XL	4	16
3XL	8	15
4XL	6	23
5XL	8	32
6XL	8	46
7XL	8	61
8 XL	16	92

Available and Planned Operations with Virtual Machines

Currently, VMWare private agent allows performing these operations with virtual machines:

- run instances,
- start and stop instances,
- reboot instances,
- terminate instances,
- discover instances existing in the VMWare infrastructure but deployed not by Maestro (discovery check is performed every 30 minutes).

In future releases, we plan to expand this list of the available operations with virtual machines:

- recover an instance from volume,
- resize an instance,
- create and attach additional volumes to the existing instances.

Other upcoming updates will include such operations as:

- push-notifications which inform that a new instance has appeared in the infrastructure,
- automatic adding of an instance to the domain,
- automatic access to the instance via the terminal,
- automatic initial configuration of the instance via the uploaded script.

These new operations will be announced and described separately.

Disaster Recovery Scenario

Current implementation of VMWare private agent supports this disaster recovery scenario:

Figure 22 – Disaster recovery scenario

It is performed via the Maestro UI and includes these simple steps:

- (preliminary) copy the applications data to a volume of the existing virtual machine,
- detach this volume from the instance,
- run a new instance,
- attach the existing volume to this new instance.

5 ON-PREMISE SOLUTION

Maestro On-Premise solution is a full functioning version of the Maestro application that can be deployed on any individual instance, without reference to any cloud provider or their datacenters.

On-Premise solution was developed as an alternative to SaaS-based cloud management solutions which are unacceptable for some enterprises due to the specifics of business or local regulations for data storage.

The difference between the existing SaaS and On-Premise solutions is that the latter is not connected to a specific cloud. A customer may not be interested in having a configured account in a certain public cloud (e.g., Amazon) due to various reasons: either a specific software was installed on an instance, or a agreement exists with a different cloud provider, or the customer has own data centers or servers). A customer may also consider that keeping information on their own servers provides better security.

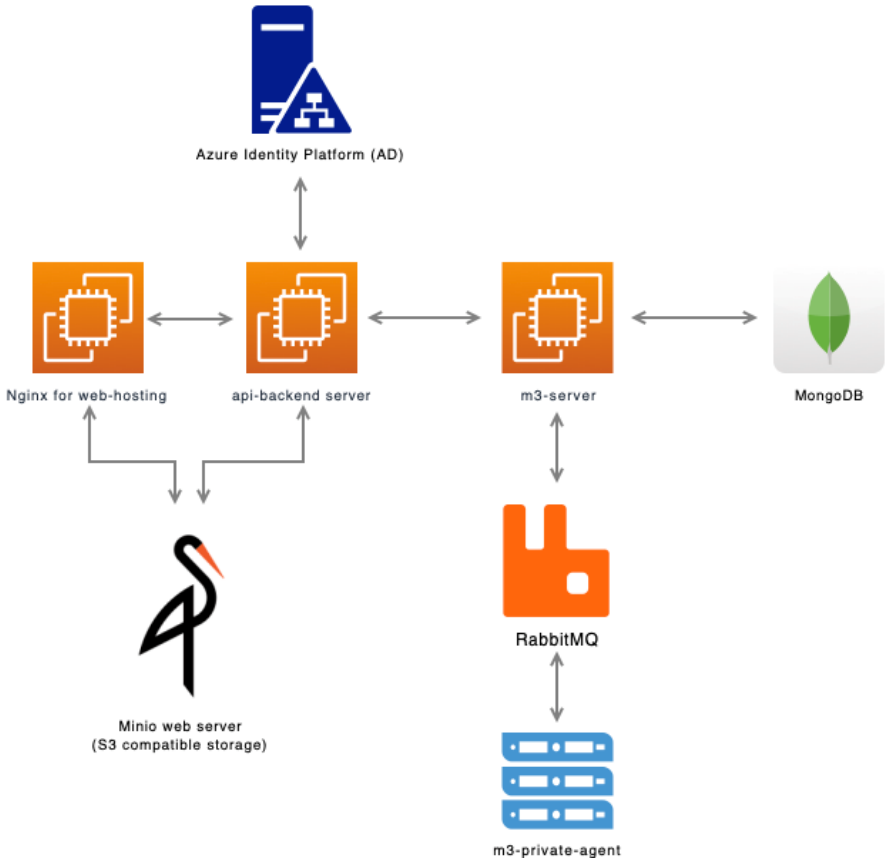


Figure 23 – On-Premise architecture scheme

On-Premise installation is designed for using open-source components such as MongoDB, RabbitMQ, etc.

Maestro On-Premise solution can be installed on lower capacities and then be expanded to larger servers and their sets. Such an approach will satisfy the needs of any customers on their infrastructure.

The development procedure for the On-Premise solution does not require a specific knowledge.

On-Premise solution supports most of the Maestro functionality: running and managing of instances, creating of images and volumes, creating and managing of SSH keys, full support of permissions and notifications functionalities, full support of billing and reporting functionalities, full support of Terraform, etc.

On-Premise solution can also support private clouds upon additional configuration.

6 CLOUD ABSTRACTION LAYER AND CLOUD INTEGRATIONS

Cloud abstraction level allows to hide the differences, derived from specifics of rendering services by various cloud providers.

Maestro is currently integrated with the following Cloud providers and platforms, and allows unified access to them:

- AWS
- Microsoft Azure
- Google Cloud Platform
- OpenStack
- VMware

The information on the steps and necessary parameters, needed to connect accounts in respective providers to Maestro is given in respective documents. Further in this section, you can find the summary for each case.

6.1 INTEGRATION WITH AWS

To connect your AWS accounts to Maestro, you need two type of operations – gathering billing data and managing virtual infrastructures. The following actions should be performed to enable billing data collection:

1. Creating a Cost & Usage report
2. Configuring AWS Athena Service
3. Creating an IAM role for billing access
4. Enabling management

6.1.1 Pre-requisites

In order for administrators of AWS accounts, intending to enable access to billing information and infrastructure management for the Maestro application, the following prerequisites should be met:

1. A user needs to have an active AWS account.
2. A user needs to have access to the account with root user permissions.
3. A user needs to know the ID of the account to which the access will be provided. Please address Maestro team for this information.

The final steps on enabling access to billing information and infrastructure management on AWS accounts will be performed by the Maestro development team upon receiving all necessary data.

6.1.2 Expected Outcome

Once the customer administration team performs all necessary steps on the customer's account, the following outcome is expected:

1. A user has a Cost and Usage report created and configured for the user's account.
2. A user has got an IAM role for enabling access to billing data of the user's account.
3. A user has got an IAM role for enabling management of infrastructure on the user's account.

4. Users have a file or several files which contain this data for further sharing with Maestro team:
    - For billing:
        - The ID of the AWS Account from which billing will be gathered
        - Athena region code (for example, eu-central-1).
        - Cost and Usage reports S3 bucket name
        - Athena Query result S3 bucket name
        - Billing Access Role ARN name
        - Glue Database name
        - Glue Table name
        - Cost and Usage reports status file S3 Path
    - For management:
        - Management Access Role ARN name
        - The ID of the AWS Account for which management via Maestro should be enabled.

6.2 INTEGRATION WITH MICROSOFT AZURE

The steps needed to enable integration with Azure depend on the subscription type used by the customer – whether it is an Enterprise Agreement, or a CSP-type.

6.2.1 Account with Azure EA subscription

To connect your AWS accounts to Maestro, you need to perform the following actions:

1. Generate an API Key for enabling billing access
2. Register application in Azure AD by:
    - Registering Maestro as an Azure AD application
    - Assigning the application to a role
    - Getting values foe signing in

6.2.1.1 Pre-requisites

In order for administrators of Azure accounts, intending to enable access to an Azure Enterprise Agreement billing, the following prerequisites should be met:

1. A user needs to have an active Azure EA subscription.
2. A user needs to have access to the subscription with the following permissions:

Task	Portal	Role
Generating an API key	https://ea.azure.com	Enterprise Enrollment Admin
Registering an application in Azure AD	https://portal.azure.com	Admin (with permissions to register an application and assign a role to it)

To generate an API key, Maestro needs requires the following pre-requisites:

- Enrollment Number
- Azure EA API key

To provide Maestro with the necessary access, you need to provide a set of access-related values:

- Directory ID (Tenant ID)
- Application ID (Client ID)
- Authentication Key

The final steps on enabling access to Azure Enterprise Agreement billing will be performed by the Maestro development team upon receiving all necessary data.

Expected Outcome

Once the customer administration team performs all necessary steps on the customer's account,, the following outcome is expected:

1. Maestro application is registered in the user's subscription as an Azure AD application with a Contributor's role.
2. The user has a file or several files, which contain the following:
    - Azure EA enrollment number
    - Azure EA enrollment API key
    - Azure EA subscription ID
    - Directory ID (Tenant ID)
    - Application ID (for Maestro application)
    - Application Authentication key (for Maestro application)

6.2.2 Account with Azure CSP subscription

To connect your Azure accounts to Maestro, you need to perform the following actions:

1. Enable access to CSP billing information by:
    - Locating the Microsoft Partner Network ID
    - Locating the default domain
    - Creating the CSP web app
    - Generating a key for the web app
2. Register application in Azure ID by:
    - Registering Maestro as an Azure AD application
    - Assigning the application to a role
    - Getting values for signing in

Pre-requisites

In order for administrators of Azure CSP subscriptions who want to enable access to the subscription billing information for the Maestro application, the following prerequisites should be met:

1. The user's organization needs to have a Microsoft Azure Partner status.
2. The user needs to have access to the Microsoft Azure Partner Center.
3. The user's organization needs to have a partner domain (can be retrieved via Microsoft Azure Partner Center).
4. The user needs to have access to the subscription with the following permissions:

Task	Portal	Role
Creating the CSP Web App	https://partnercenter.microsoft.com/	Global Admin
Registering an application in Azure CSP	https://portal.azure.com	Admin (with permissions to register an application and assign a role to it)

The final steps on enabling access to Azure CSP subscriptions billing will be performed by the Maestro development team upon receiving all necessary data.

Expected Outcome

Once the customer administration team performs all necessary steps on the customer's account,, the following outcome is expected:

1. A CSP Web app is created.
2. Maestro application is registered in the user's subscription as an Azure AD application with a Contributor's role.
3. The user has a file or several files, which contain the following:
    - MPN ID
    - Default Domain name
    - CSP web app Application ID
    - CSP web app key
    - Directory ID (Tenant ID)
    - Application ID (for Maestro application)
- Application Authentication key (for Maestro application)

6.3 INTEGRATION WITH GOOGLE CLOUD PLATFORM

To connect your Google accounts to Maestro, you need two type of operations – to enable access to Google Account billing and provide an opportunity to manage virtual resources. The following actions should be performed to set up billing:

1. Activate billing export in Maestro by:
    - Creating a new project (optional)
    - Creating a billing account
    - Linking a billing account to a project
    - Activating billing export
2. Create service account

To enable project management for Maestro the following actions should be made:

1. Configure Google Project for Maestro
2. Create Terraform service account

6.3.1 Pre-requisites

In order for administrators of Google accounts, intending to enable access to billing information and infrastructure management for Maestro application, the following prerequisites should be met:

1. A user needs to have an active Google account.
2. A user needs to have access to the account with admin permissions.
3. Billing data export to BigQuery should be enabled.

The final steps on enabling access to billing information and infrastructure management on Google accounts will be performed by the Maestro development team upon receiving all necessary data.

6.3.2 Expected Outcome

Once the customer administration team performs all necessary steps on the customer's account,, the following outcome is expected:

3. Maestro service account is registered in the user's Google account.
4. A user gets a file or several files, which contain the following:
    - Project ID
    - For Billing:
        - Billing Account ID
        - BigQuery Dataset ID
        - BigQuery Table ID
        - Billing Service Account credentials in JSON format
    - For Management:
        - OAuth2 Client ID
        - OAuth2 Client Secret
        - OAuth2 Refresh token
        - OAuth2 Access token
        - Service Account email for Terraform

6.4 INTEGRATION WITH OPENSTACK

To connect your OpenStack accounts to Maestro, you need to set up and configure a private agent in OpenStack. The following actions should be performed:

1. Install private agent
2. Install M3Admin
3. Configure private agent

6.4.1 Pre-Requisites

In order for administrators of OpenStack accounts, intending to enable access to their infrastructure management for Maestro application, the following prerequisites should be met:

This data can be obtained upon downloading the rc.sh file from the OpenStack Horizon via the export config option. This file contains the most part of information which is required for connecting to OpenStack, such as:

- Open Stack Host
- Project ID
- Project Name
- Project User ID
- Project User Name
- Project User Password
- Admin User ID (optional)
- OpenStack RabbitMQ (optional)
- OpenStack RabbitMQ User Name (optional)
- OpenStack RabbitMQ User Password (optional)
- OpenStack RabbitMQ Hostname (optional)
- OpenStack RabbitMQ Exchanges Name (optional)
- Admin User Name (optional)
- Admin User Password (optional)
- Security Group Name
- Image IDs
- Image Names
- Flavor IDs
- Flavor Names
- Public Image Owner ID
- Network ID
- Network Name
- Security Group ID

The final steps on enabling access to infrastructure management on OpenStack accounts will be performed by the Maestro development team upon receiving all necessary data.

6.4.2 Expected Outcome

Once the customer administration team performs all necessary steps on the customer's account, users will get a configured private agent which is ready to be used for managing a specified OpenStack region.

The following information about the created entities should be returned to the Maestro team:

- Image Aliases Names
- Region Names
- Tenant Names
- Shape Names

7 EVENT-DRIVEN ARCHITECTURE

Maestro is developed based on the following principle: every action is an event and a result of this event. If a user wants to perform an action, he/she notifies the system about it, and the intent becomes an event. Then the event is processed, and the desired action is performed on the basis of the request.

7.1 EVENTS AUDIT

In the Maestro system, all user's actions get through an audit. If, for example, a user has run or stopped an instance, a record about this event is saved to the database.

The information about any action performed by users can be obtained in the form of an event audit and based on the latter the billing is calculated.

Altogether, the audit is a core of the Maestro application, as at first there is an action, then the action audit is performed and then based on the audit all necessary post-actions are done. For example:

1. a user runs an instance;
2. information about this occurred event appears in the system;
3. the audit of this event is performed;
4. audit record appears in the **Content View** – **Audit** tab;
5. the handler responsible for sending audit notifications sends a letter informing that the instance was run;
6. based on the audit counters are changed, quotas are recalculated, etc.

Thus, the audit is one of the core mechanisms of the Maestro application.

7.2 EVENT-DRIVEN ARCHITECTURE

The basic principle of building the Maestro framework is the following: any action triggers an event. Each subsequent state of the database can be restored by events: if we backup the database at some point in time, and then gradually roll up the changes due to event processing, then, while reconstructing the events in the order in which they arrived, we will restore the database state, at the moment when it fell.

Maestro uses RabbitMQ and AMQP as an event broker for an event-driven approach. RabbitMQ is used as one of the mechanisms for listening to an event, since events can be generated by both lambdas and the private agent.

If the private agent generates an event, it sends it to RabbitMQ, the system listens to the event and processes it, i.e. all events related to instances are stored there.

If a Lambda generates an event, then it sends it to the server directly via HTTP.

The Maestro system uses SDK as a single point through which events are transferred to the Maestro application. But there are gaps, considering that if we throw event data through HTTP, and the server is unavailable at this time, it can be lost. RabbitMQ event data to accumulate and is the layer that provides persistency and consistency of events.

.

8 M3 SDK

Maestro SaaS Software Developer's Kit (SDK) has been implemented using JSON-RPC as a Remote Procedure Call (RPC) protocol. It uses JSON for serialization and has been chosen due to its simplicity, lightweight and cleanness, among other advantages (see below).

The protocol works by sending a request to a server implementing it. The client is typically a software application, intended to call a single method of a remote system. Multiple input parameters can be passed to a remote method as an array or an object, whereas the method itself can return multiple output data as well.

A remote method is invoked by sending a request to a remote service using HTTP or a TCP/IP socket (starting from version 2.0). When using HTTP, the content-type can be defined as application/json.

All transfer types are single objects, serialized using JSON. A request is a call to a specific method provided by a remote system. It must contain three certain properties:

- **method** – a string with the name of the method to be invoked
- **params** – an array of objects to be passed as parameters to the defined method
- **id** – a value of any type, which is used to match the response with the request it is replying to.

The receiver of the request must reply with a valid response to all received requests. A response must contain the properties mentioned below.

- **result** – the data returned by the invoked method. If an error occurred while invoking the method, this value must be null.
- **error** – a specified Error code if there was an error invoking the method, otherwise null.
- **id** – the id of the request it is responding to.

Since there are situations where no response is needed or even desired, notifications were introduced. A notification is similar to a request except for the id, which is not needed because no response will be returned. In this case the id property should be omitted or be null.

There are many different ways to implement API for the application. For example, REST based API means that each unique URL is a representation of some object or resource. A user can get the contents of that object using an **HTTP GET** method, to delete it, then use **POST**, **PUT**, or **DELETE** to modify the object (in practice most of the services use **POST** for this). **REST** is great for public-facing APIs, intended for use by other developers. They can be designed in accordance with common standards, as to not require a lot of preexisting knowledge about the service that is going to be used.

However, in our case EPAM Cloud Orchestration API has to provide access to a specific set of functions.

Our main goal was to develop a lightweight, well-specified interface that does not have direct access to data but performs a remote call of Orchestration functionality.

We have considered the following advantages of *JSON-RPC*:

- **Unicode** – both *JSON* and *JSON-RPC* support *Unicode* out-of-the-box,
- **Transport-independent** – *JSON-RPC* can be used with any transport socket: TCP/IP, HTTP, HTTPS etc.,
- **Direct support of Null/None**,
- **Support of named/keyword parameters**,
- **Built-in request-response matching** ("id"-field),

- **Error codes**: ranked and well specified, covering a wide spectrum of possible exceptions.
- **Notifications**.

All procedure calls are strictly atomic and return a well specified, determined result. Clients are not required to know procedure names and the specific order of arguments, because the specifics of JSON-RPC is hidden within our implementation to make the SDK more convenient for use.

8.1 CONFIGURATION

8.1.1 Maven Configuration for Maestro JAVA SDK

Add the following within the <dependencies> section of your POM:

```
    <dependencies>
         <dependency>
            <groupId>com.m3.sdk</groupId>
            <artifactId>m3-sdk</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
```

The artifacts are available via EPAM Maven Repository.

```
<repositories>
        <repository>
          <id>artifactory.epam.com</id>
          <name>artifactory.epam.com-releases</name>
          <url>http://artifactory.epam.com/artifactory/EPM-CIT</url>
        </repository>
    </repositories>
```

8.1.2 Entry Point

The entry point for Maestro Java SDK is **M3Sdk**. It can be reached by the following address: **com.m3.sdk.**

8.1.3 Typical Working Scenario

1. Create a client

```
IM3client client =
M3Sdk.client(serverContextProvider, clientContextProvider, credentialsProvider, access
KeyProvider);
```

2. Execute command

```
M3ApiResult result = client.<commandName>(
                "parameter1",
                "parameter2",
                 …);
```

8.1.4 Maestro SDK Structure

The following paragraph describes the structure of Maestro SDK, which is available in the project repo.

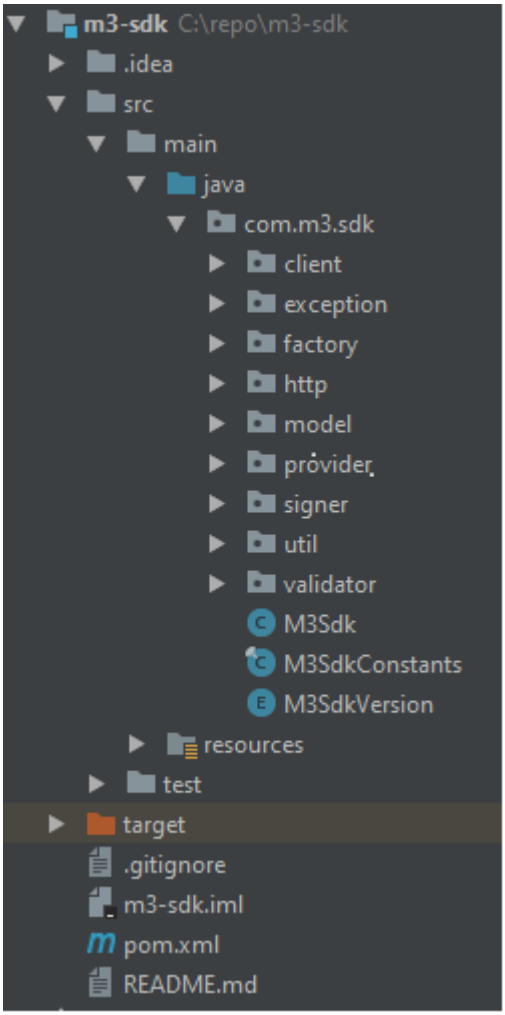


Figure 24 – Maestro Java SDK structure

8.2 AUTHORIZATION ALGORITHM

The SDK client for Maestro API uses three headers to provide authorization information into the server.

- **Maestro-Authentication** - the request signature generated by M3 SDK based on Maestro-accessKey provided by SDK client.
- **Maestro-request-identifier** - the client identifier provided by the SDK Client and used for the security purpose.
- **Maestro-accessKey** - the identity that should be registered in EC2 parameters store and used to verify the client. To register the identity, please address the M3 admin team.

The authorization is performed according to the following scenario:

1. Get the long representation of the request date from <Maestro-date> header:

```
long date = new Date().getTime();
```

2. Construct SecretKeySpec from the secretKey and date

```
byte[] bytes = Charset.forName(“UTF-
8”).encode(String.format(“%s%s”, secretKey, date)).array();
SecretKeySpec secretKeySpec = new SecretKeySpec(bytes, “HmacSHA256”);
```

3. Generate the sign:

```
Mac mac = Mac.getInstance(“HmacSHA256”);
Mac.init(secretKeySpec);
byte[] message = mac.doFinal(String.format(“M3-
POST:%s:%s”, accessKey,date).getBytes());

StringBuilder builder = new StringBuilder();
for (final byte element : message) {
  builder.append(Integer.toString((element & 0xff) + 0x100, 16));
}
String sign = builder.toString();
```

4. The new String(sign) is <Maestro-Authentication> header

On the Server side, it is needed to get the same signature and compare it with the Client’s signature.

Java SDK sample:

If you use Maestro Java SDK, all of the above is performed inside M3Client class.

```
M3Sdk.client(new M3StaticServerContextProvider("serverUrl"),
new M3StaticClientContextProvider(new M3ClientContext("clientIdentifier")),
   new M3StaticCredentialsProvider("accessKey","secretKey"),
new M3StaticAccessKeyProvider("accessKey"))
```

Also, you can replace static providers with your own implementations of IM3ServerContextProvider, IM3ClientContextProvider, etc.

9 DYNAMIC UI

Dynamic UI is an UI which depends on user's state and encourages users to interact with it. Maestro contains a set of wizards that allows users to perform action in a user-friendly way.

Wizards contain various display components: checkboxes, radio buttons, controls, text areas. For each of these elements a correspondent UI component is created. The implemented logic allows to directly define the behavior of the component as autonomous element (display, font, etc.), as well as describe the interaction with other components.

The logic is described in the JSON script, which is provided by the backend. The script is based on a specific pattern containing description of components to be displayed on a page, and the parameters of how the components should interact with each other. The application UI will be displayed according to this logic.

Since wizards are the main unit of management in our application, it is sometimes required to create more wizards when implementing new features. In terms of Maestro solution such wizards are implemented based on the already created logic, as a result having no need to additionally bring UI specialists. We created a process which allows to dynamically configure UI based on pre-written logic, thus sufficiently speeding up the time for development.

Using this approach allows to cut down the development time and implement the localization tasks. Dynamic UI also allows to use the UI component, which was created once, both in native and web applications.

9.1 ANGULAR 14

Angular is a framework, which imposes certain conditions to its architecture. Unlike a library that allows you to write in any style, the framework obliges you to follow certain rules.

It is also very suitable to use Angular in large projects, since the architecture there is clearly visible, the project components and services are defined, as well as the processed of interaction with the backend and authentication are highlighted. With Angular 14, it's much easier to manipulate a large project.

9.2 NATIVE SCRIPT

The Native Script is based on Angular, i.e. there is no need to change the development language. The same developer can contribute to both the web application and the mobile application. It is possible to use the existing logic, i.e. reuse existing solutions from the web to mobile. Since Maestro is a hybrid solution, one code is written, upon which, at the end after the build, two bundles will be obtained: for IOS and Android. Thus, such a development flow sufficiently reduces time for reduces development without involving additional specialists.

10 MAESTRO DATABASES

Maestro uses two databases to process its datasets. The first is MongoDB, the second one is DynamoDB. MongoDB is a document-oriented NoSQL database program. DynamoDB is a serverless database provided by AWS. Compared to MongoDB, DynamoDB provides faster read and write speeds, but it is significantly more expensive and can store lesser amounts of data. At the same time, Maestro required large volumes of data for billing, but speed is not sufficient,

Thus, Maestro uses MongoDB to store data. Thanks to MongoDB, Maestro database structure can be adjusted to various business requirements, which cannot be done with standard databases.

MongoDB is located on Maestro servers. Maestro fully manages it as well as installs and monitors server clusters. Its free structure allows adapting models to the necessary business requirements, i.e. in SQL databases the structure should be designed at the stage of functionality development.

10.1 MONGODB ADVANTAGES

MongoDB has a set of advantages which was made it a solution of choice:

- Allows storing unstructured data of a quite a large maximum volume up to 16MB per one document.
- Allows make ad-hoc queries, although indexed queries are processed much faster than queries without indexes.
- Allows using nested queries.
- Low cost.

10.2 MONGODB CONSTRAINS

Unlike traditional SQL databases, it does not have the Join function. To query two or more tables, you can do the following:

- perform either several queries on the client side (i.e. on the application server) OR
- use the aggregation framework (such a function is Available in MongoDB).

MongoDB makes several queries and then returns the result in a specific way. Maestro uses such requests for billing.

Absence of transactions in MongoDB does not influence Maestro functionality as it is a distributed application, and requests to the database come simultaneously from different servers.

10.3 MONGODB USAGE IN MAESTRO

Maestro mainly focuses on using MongoDB for billing procedures. Besides, other parts of the application use capped collections functionality which enables flexibility and the ability to store large amounts of data.

Unlike some other databases, MongoDB requires approximately 1GB of RAM per 100.000 assets. If the system has to start swapping memory to disk, this will have a severely negative impact on performance.

MongoDB is **not very fault tolerant**, but in this case a cluster can be a solution. All environments of Maestro use clusters of this database, represented by several running instances. One of them is appointed as a master and all writing operations are performed through it. All other databases are used as reading replicas.

At the same time, these replicas can be used to create backups of the databases at any time without losing application performance.

Besides, MongoDB supports JavaScript natively and can process queries written in JavaScript. Sometimes, it is also used for complex scripts, such as database patches.

11 SUPPORT AND DEVOPS TOOLS

11.1 TERRAFORM AND TERRAFORM PROVIDER

The main IaC toos taken on board by Maestro is Terraform by HashiCorp – a cross-platform solution which allows managing complex infrastructures hosted in multiple clouds. Terraform is a tool for building, changing, and versioning infrastructure in a safe and efficient manner.

Terraform is built on a plugin-based architecture, enabling developers to extend Terraform by writing new plugins or compiling modified versions of existing ones.

The main principle of this tool is in Infrastructure as a Code approach, i.e. infrastructure parameters are described in a template, in a JSON and HCL formats. Both formats are supported by Maestro.

11.1.1 Integration with Terraform

Maestro integration with Terraform provides its users with ability to:

- store templates in AWS S3 and in GitHub and upload templates from the repository
- automatically react to updates in GitHub (auto plan and auto apply)
- plan and apply templates
- review and validate logs
- lock templates for further modification, planning or applying
- share and re-use templates

M3 supports the following Terraform functionality:

- ability to upload templates through the UI
- possibility to edit templates via Maestro UI, it is possible to receive information about the resources that were deployed and monitor the logic of the template
- imposition of limitations for specific users, i.e. for users and for regions.

.

To register the GitHub based template on Maestro, the following parameters should be specified:

- **source type** (GitHub) – only after this, the necessary input field will appear,
- your GitHub **username** and **password**,
- **URI** of the relevant GitHub repository and its **branch**,
- **folder** where your Terraform templates are stored,
- (optional) Terraform variables and their values to be used as default ones when the template is applied.

You must also select what actions Maestro will take once the template is updated in the GitHub repository (None, Auto plan, or Auto apply):

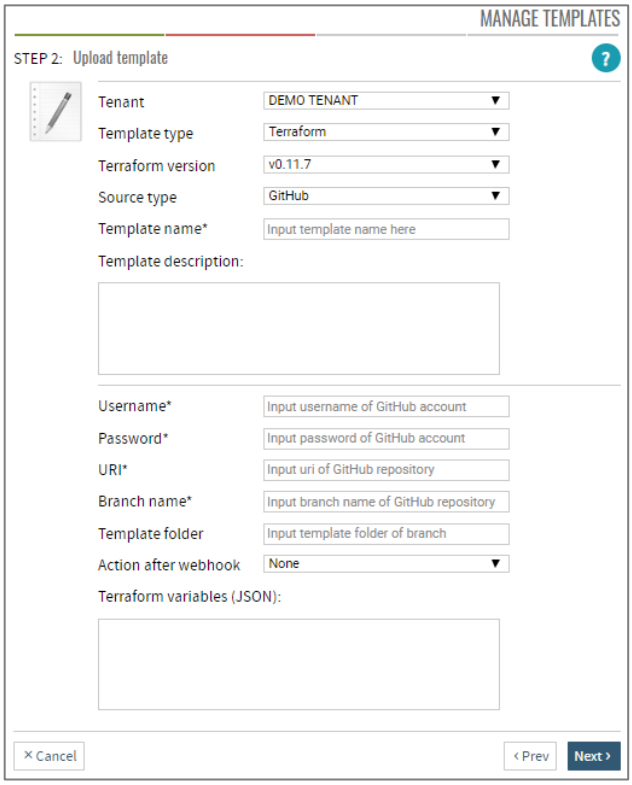


Figure 25 – Upload templates window

Once registered, the both GitHub and AWS-based Terraform templates can be reviewed in the **Catalog** page with any version of Maestro Orchestrator. Terraform templates stored in GitHub are updated and validated automatically once they are changed in the repository – you do not need to do it specifically for Maestro.

In the Catalog page, you can review and manage all the existing templates (both Terraform and CloudFormation ones, supported by AWS):

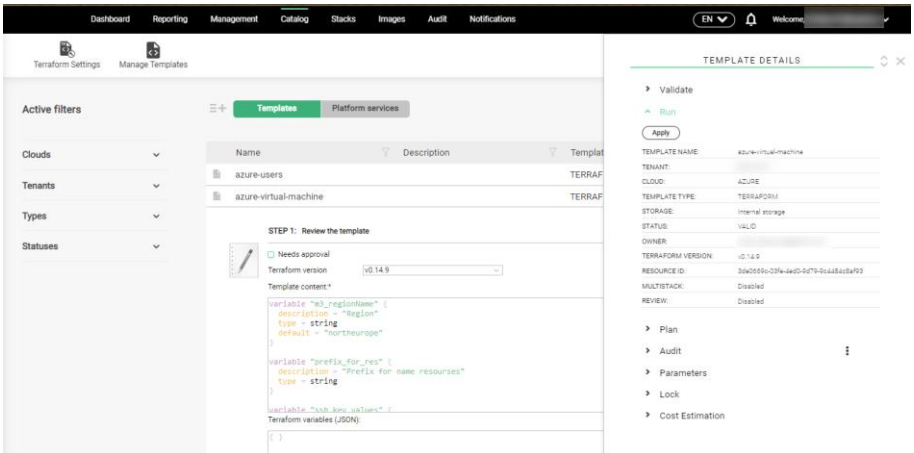


Figure 26 – Service catalog page

Provided information:

- name and description of the Terraform template, its type and status
- template code
- validation status and validation logs
- events related to the template
- default values of the template parameters

.

Terraform templates management is performed with a set of wizards and buttons, available on the **Dashboard** and **Catalog Page**. The table below lists the available stack templates actions and their basic details:

Action	Description	Action Source
Upload	Upload a stack template to Maestro	Manage Templates Wizard
Plan	Create execution plan and estimate the results of template application.	Catalog Page
Apply	Apply the stack template on infrastructure	Catalog Page
Remove	Deletes the stack template from Maestro	Manage Templates Wizard
Lock	Locks the template from other users, so that they cannot modify or remove it	Catalog Page
View execution log	Display the execution logs of the template	Catalog Page
Download logs	Download template execution or planning logs in a zip archive	Catalog Page

Terraform allows to automate all changed and provide versioning of the infrastructure.

Terraform CLI is a free tool and can be used by command line utility.

11.1.2 Terraform Provider for Maestro

Terraform is a framework for configuration, but one Terraform template cannot be written in a unified way so that it could be used for different cloud providers (the template for AWS will not work if you deploy it in Azure). Maestro provides its own custom tool - Terraform-Provider that gives the users possibility to work with the provided API so that they can deploy the infrastructure with a single template on any cloud provider supported by Maestro, including private OpenStack region.

Working with Terraform Provider for OpenStack

Requirements:

The following software should be installed before you use Terraform:

- Terraform 0.12+.
- Go 1.13 (to build the provider plugin).

To start using the provider, do the following:

1. To build the Terraform Provider plugin, run:

```
#linux
go build -o terraform-provider-m3_v0.2
#windows
go build -o terraform-provider-m3_v0.2.exe
```

2. Move the plugin to the user plugins directory (you can find more details here):

```
#linux
mv terraform-provider-m3_v0.2 ~/.terraform.d/plugins
#windows
move terraform-provider-m3_v0.2.exe %APPDATA%\terraform.d\plugins
```

3. Specify the provider settings:

```
provider "m3" {
      url = "http://ip:port/maestro/api/V3"
      access_key = "access_key"
      secret_key = "secret_key"
      user_identifier = "user_identifier"
}
```

The current implementation supports the limited scope of operations, but the existing configuration is enough to run a VM and create an image.

Below, you can find an example of the provider usage by one of our colleagues who contributed the feature:

These parameters are enough to run an instance:

```
resource "m3_instance" "my-server" {
      image = "CentOS7_64-bit"
      instance_name = "test_name"
      region = "EPAM-OPENSTACK-3"
      tenant_name = "EPMC-EOOS"
      shape = "MINI"
      key_name = "sshkey"
}
```

These parameters are enough to create an image:

```
resource "m3_image" "my-image" {
      tenant_name = "EPMC-EOOS"
      region_name = "EPAM-OPENSTACK-3"
      image_name = "ImageFromTf"
      source_instance_id = "ecs00100019F"
      description = "Here is image desccription"
}
```

11.2CHEF

Chef is an open source automation tool that allows to quickly set up and configure the necessary infrastructure, and to change the configuration if needed. System configuration files that describe how Chef manages server applications are called **recipes**. Several recipes grouped together constitute a **cookbook**.

Currently this functionality is implemented and working in a test mode on the AWS cloud, supporting Jenkins and Artifactory automation systems.

Functionality was tested on AWS cloud using the following parameters:

OS	Shape	Role
CentOs7	Medium	artifactory-acs
Ubuntu16	Medium	jenkins_epc

All profiles have some requirements to instance OS/Shape.
In the current implementation, validation for these requirements is not supported, so some combination for OS/Profile/Shape may not be working.

11.2.1 Configuring Chef

See an example of configuring Chef in Maestro below:

1. A region should be activated with parameters, set specifically for working with Chef. To get an access to such a region or to configure Chef for the already activated project, send a specified request to the Maestro support team.

   Thus, when launching an instance, this region should be specified in a **Region** field together with other parameters for running the instance.

2. Select the **Set chef configuration** checkbox and specify the **Chef Profile**. The profile defines the configuration that should be installed on the instance.
3. On the screen you can review information about the Chef Profile, which was specified on a previous step.

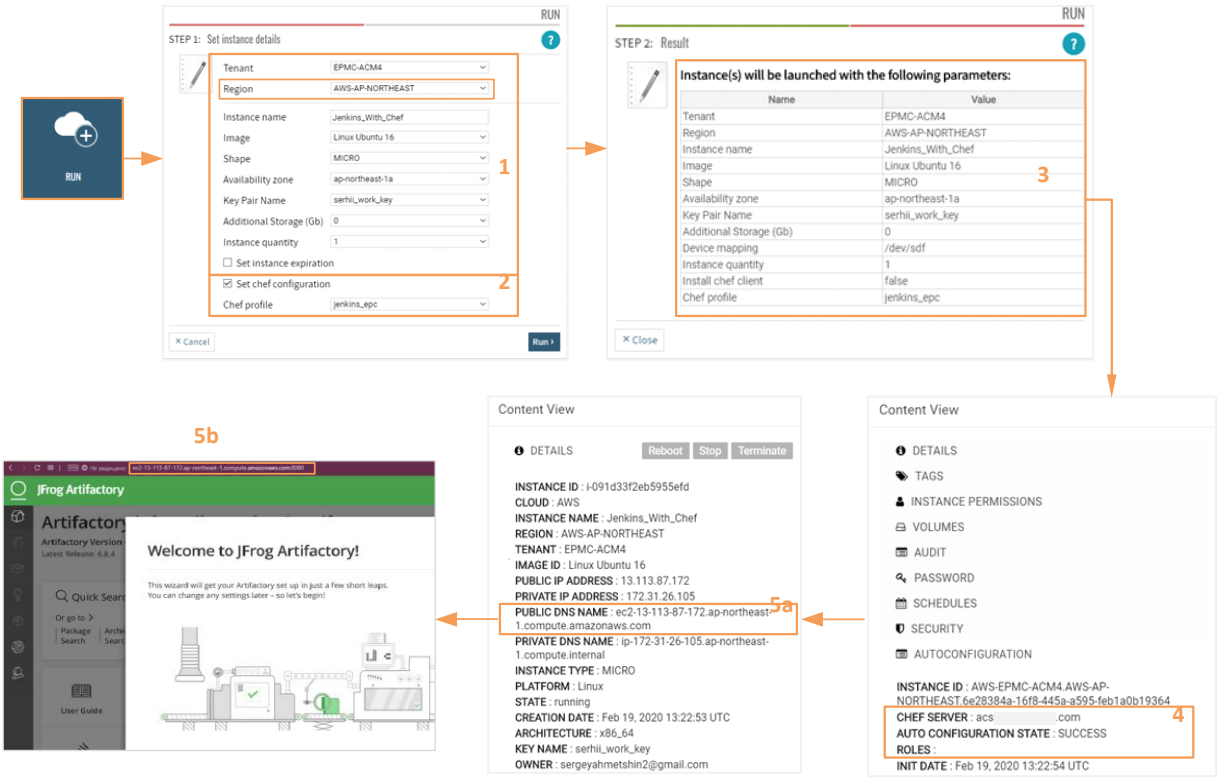


Figure 27 – Chef configuration example

4. The **Autoconfiguration** tab is added to the **Content View** section of the specified instance in AWS cloud. It contains useful information about chef role (not available at the moment), state and chef server. As a result, the installation of the software selected depending on the Chef Profile was successfully performed on the instance.
5. In order to access and interact with the interfaces of the installed software (in this case - artifactory and Jenkins), perform the following actions:
    a. Copy the public DNS name of the instance;
    b. Open the User Interface of the launched application, using the copied DNS name:
        - For Jenkins: <DNS_NAME>:8080
        - For artifactory: < DNS_NAME > :8081/artifactory.

11.2.2 Work Principles

When a software is being installed on an instance, an init script is added to it. The instance gets registered in Maestro. After that, communication between the Chef server, Maestro server, and a Chef client is established.

See the scheme for an init script used by Chef below:

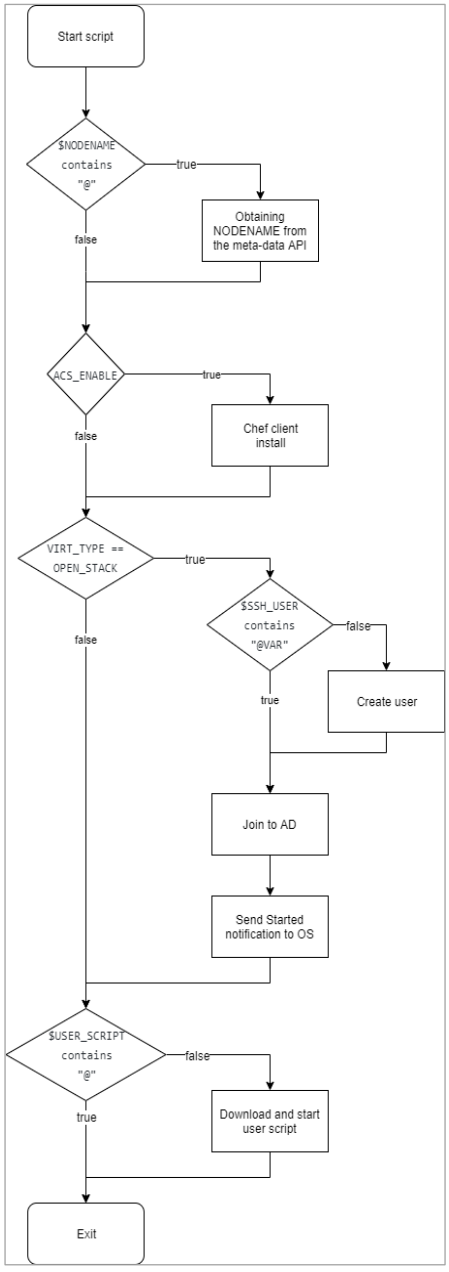


Figure 28 – Init script used by Chef

The table below lists required variables for the init script.

Variable	Placeholder	Description	Example
CONF_URL	@VAR_EP_ORCH_IP	URL endpoint to Orchestrator	https://config.cloud.epam.com/orchestration-dev
orch_url	@VAR_CONFIG_URL	URL endpoint to Orchestrator (when VAR_ACS_ENABLE=false)	https://qa.cloud.epam.com
VIRT_TYPE	@VAR_VIRT_TYPE	Virtualization type	OPEN_STACK/AZURE/AWS/GOOGLE
NODENAME	@VAR_NODENAME	instanceId	ecs00012345678
ACS_ENABLE	@VAR_ACS_ENABLE	autoConfigurationDisabledType from Projects	true/false
CHEF_PROJECT	@VAR_PROJECT_CHEF	Dedicated Chef server is used	true/false
CHEF_SRV	@VAR_CHEF_SERVER	Chef server fqdn	acs.cloud.epam.com
CHEF_ENV	@VAR_CHEF_ENV	Chef environment	development/production
CHEF_ORG	@VAR_CHEF_ORG_NAME	Chef organization name	epam-dev/epam-qa/epam
SSH_USER	@VAR_SSH_USER	Default OS username	ubuntu/admin/centos
USER_SCRIPT	@VAR_USER_SCRIPT	String containing links to user scripts and launch parameters, if used	/api/files/0aca25d4-ffc8-46c0-907a-0dc4f21cd98a/user-init.sh:fuu#bar
NOTIF_URL	@VAR_NOTIF_URL	Used as a signal that the VM has entered the running condition	https://qa.cloud.epam.com/api/openstack/notification/running
OS_CHECKSUM	@VAR_NOTIF_CHECKSUM	Used together with NOTIF_URL	SXVZSjl0czlJbENFSmxpdXFnSk0wUlBo

11.3 ANSIBLE

As an automation system for provisioning and configuring infrastructure, Ansible allows installing software on virtual machines and manage large clusters of them on different cloud providers.

A distinctive feature of Ansible is that the system has a server, which does not contain the client side, i.e. you do not need a client to be installed on a virtual machine. Ansible accesses virtual machines via SSH, and is intended for working with Linux machines, though it is possible to use it for Windows as well. Thus, a Linux server has access to a virtual machine via SSH and performs pre-defined tasks there.

The tasks which should be performed on a virtual machine are described in playbooks. A playbook is structured on lambda, which describe conditions of the necessary tasks (similar to a template or a stack).

Ansible also works with **host groups** where hosts are described/registered. The host group defines virtual machines that Ansible should affect. Grouping is necessary to define the scope of work, i.e. a playbook is launched for a certain group.

Every host group is created **manually** in a JSON format and contains the following:

- name of the host group
- DNS names or IP addresses of the VMs to perform automation configuration on.

For example, to install a new Java software version on a group of Jenkins slaves, you can define the conditions in a playbook. Running a playbook will make Ansible go to all hosts and perform the same playbook on them.

Dynamic Inventory

Besides, Ansible includes the **Dynamic Inventory** feature that enables retrieving information from dynamic sources. In this case, dynamic groups are created on the fly using cloud providers (Maestro). You can configure dynamic groups to any criteria - keys, security groups, tags.

Maestro works with dynamic groups, their management, and provisioning of these groups to users.

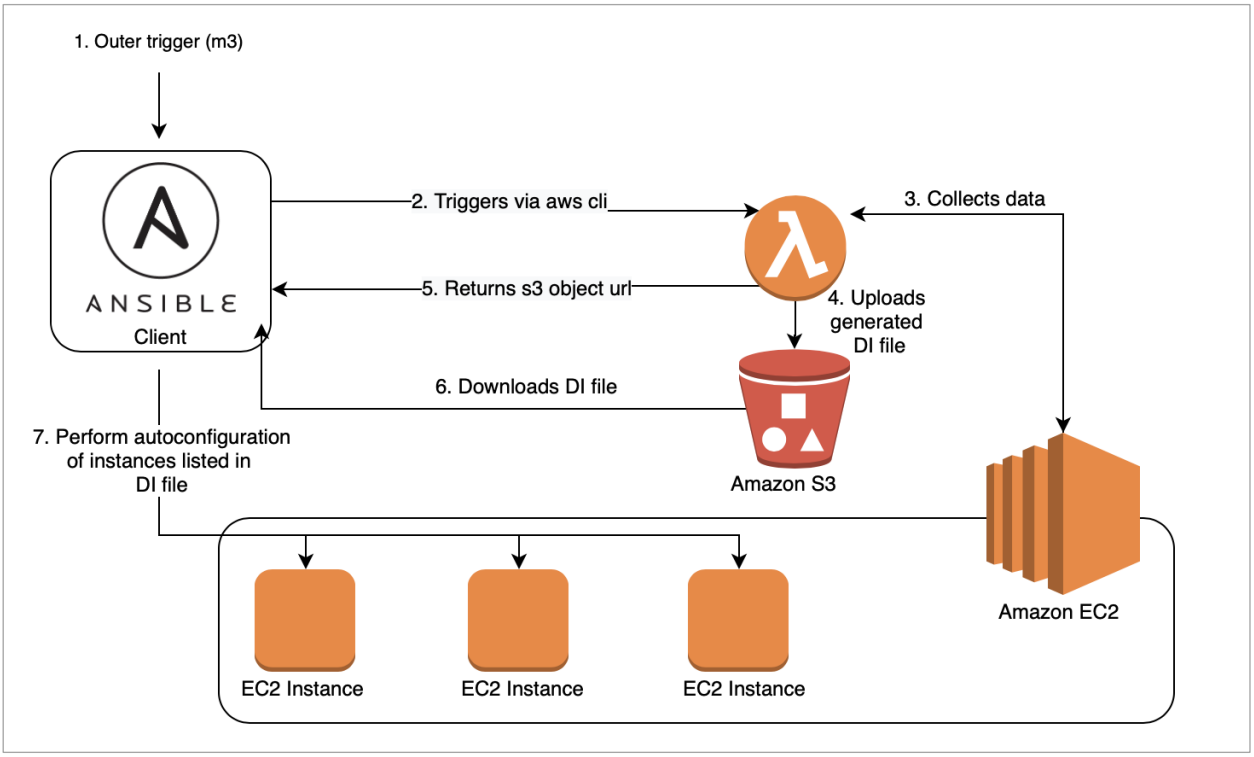


Figure 29 – Ansible and Dynamic Inventory flow

In Maestro solution, Ansible is introduced via the respective wizard on the Main Dashboard.

The wizard allows to generate an Ansible client for the selected region and tenant. The result will be provided as a downloadable zip, containing the dynamic inventory files.

To start working with Ansible client, select Ansible wizard, and perform the required steps. For a detailed information, please see Annex B.

After you perform the necessary steps, you obtain a downloadable .zip file. The zip file contains a configuration file and a Python script with lambda that configures a dynamic inventory file.

The **dynamic inventory file** is a list of virtual machines, including their IPs, DNS addresses and all other VM properties grouped by parameters, e.g. tags, keys, owners, security groups etc.

To make the Python script work, correct it according to your parameters. After that, launch the script using CLI. It will start sending the parameters, created during the generation of this Ansible client, to lambda.

Knowing the parameters, lambda describes virtual instances in a specified tenant and region. Lambda sends a request to a TCP service for obtaining a list of all VMs. Upon obtaining the data, lambda groups it according to the specified keys, loads the dynamic inventory file into S3 and send a link to this file to Ansible Client.

Resign URL, a temporary link to the file stored in S3 is signed with the credentials of this lambda and expires after 15 minutes on creation. If it is not obtained by the client within 15 minutes, the file becomes unavailable, as the infrastructure is updated every 15 minutes.

Upon loading the file, Ansible starts to analyze the file and defines what software should be configured on which VMs. Information about the software is obtained from the playbook, the list of VMs to be configured is obtained from the Dynamic Inventory file.

The list of VMs with properties grouped by virtual keys, is provided to Ansible.

Ansible is the most popular open source automation tool on GitHub. It is an automation tool used to configure systems, deploy software, and orchestrate more advanced tasks such as continuous deployments or zero downtime rolling updates.

ANNEX A – MAESTRO SAAS PERMISSIONS

EO_ORCHESTRATOR

```
AWS Managed policies
'arn:aws:iam::aws:policy/AdministratorAccess'

Custom policies
- AdministratorAccess

AdministratorAccess
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "*",
            "Resource": "*"
        }
    ]
}
```

EO_INSTANCE

```
AWS Managed policies
'arn:aws:iam::aws:policy/AmazonS3FullAccess'
'arn:aws:iam::aws:policy/CloudWatchFullAccess'
'arn:aws:iam::aws:policy/AmazonSSMFullAccess'
'arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess'
'arn:aws:iam::aws:policy/AmazonSNSFullAccess'
'arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore'
```

EO_API

```
AWS Managed Policies
'arn:aws:iam::aws:policy/AmazonS3FullAccess'
'arn:aws:iam::aws:policy/CloudWatchFullAccess'
'arn:aws:iam::aws:policy/AmazonSSMFullAccess'
'arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess'
'arn:aws:iam::aws:policy/AmazonSNSFullAccess'
'arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore'
```

EO_LAMBDA

```
AWS Managed Policies
'arn:aws:iam::aws:policy/AmazonS3FullAccess'
'arn:aws:iam::aws:policy/CloudWatchFullAccess'
'arn:aws:iam::aws:policy/AmazonSSMFullAccess'
'arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess'
'arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore'
'arn:aws:iam::aws:policy/service-role/AWSLambdaSQSQueueExecutionRole'
'arn:aws:iam::aws:policy/AmazonCognitoReadOnly'
```

```
Custom policy
- eo_lambda_policy


eo_lambda_policy
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "ec2:*",
                "elasticloadbalancing:*",
                "cloudwatch:*",
                "autoscaling:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonEC2FullAccess1"
        },
        {
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": [
                        "autoscaling.amazonaws.com",
                        "ec2scheduled.amazonaws.com",
                        "elasticloadbalancing.amazonaws.com",
                        "spot.amazonaws.com",
                        "spotfleet.amazonaws.com",
                        "transitgateway.amazonaws.com"
                    ]
                }
            },
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonEC2FullAccess2"
        },
        {
            "Action": [
                "cloudformation:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AWSCloudFormationFullAccess"
        },
        {
            "Action": [
                "sqs:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonSQSFullAccess"
        },
        {
```

```
            "Action": [
                "sns:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonSNSFullAccess"
        },
        {
            "Action": [
                "xray:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AWSXrayFullAccess"
        },
        {
            "Action": [
                "states:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AWSStepFunctionsFullAccess"
        },
        {
            "Action": [
                "support:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AWSSupportAccess"
        },
        {
            "Action": [
                "inspector:*",
                "ec2:DescribeInstances",
                "ec2:DescribeTags",
                "sns:ListTopics",
                "events:DescribeRule",
                "events:ListRuleNamesByTarget"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonInspectorFullAccess1"
        },
        {
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "inspector.amazonaws.com"
                    ]
                }
            },
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonInspectorFullAccess2"
```

```
        },
        {
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": [
                        "inspector.amazonaws.com"
                    ]
                }
            },
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": "arn:aws:iam::*:role/aws-service-
role/inspector.amazonaws.com/AWSServiceRoleForAmazonInspector",
            "Effect": "Allow",
            "Sid": "AmazonInspectorFullAccess3"
        },
        {
            "Action": [
                "kms:DescribeKey",
                "kms:ListAliases",
                "kms:ListKeys",
                "workspaces:CreateTags",
                "workspaces:CreateWorkspaceImage",
                "workspaces:CreateWorkspaces",
                "workspaces:CreateStandbyWorkspaces",
                "workspaces:DeleteTags",
                "workspaces:DescribeTags",
                "workspaces:DescribeWorkspaceBundles",
                "workspaces:DescribeWorkspaceDirectories",
                "workspaces:DescribeWorkspaces",
                "workspaces:DescribeWorkspacesConnectionStatus",
                "workspaces:ModifyCertificateBasedAuthProperties",
                "workspaces:ModifySamlProperties",
                "workspaces:ModifyWorkspaceProperties",
                "workspaces:RebootWorkspaces",
                "workspaces:RebuildWorkspaces",
                "workspaces:StartWorkspaces",
                "workspaces:StopWorkspaces",
                "workspaces:TerminateWorkspaces"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonWorkSpacesAdmin"
        },
        {
            "Action": [
                "iam:GenerateCredentialReport",
                "iam:GenerateServiceLastAccessedDetails",
                "iam:Get*",
                "iam:List*",
                "iam:SimulateCustomPolicy",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "IAMReadOnlyAccess"
        },
```

```
        {
            "Action": [
                "aws-portal:ViewBilling",
                "budgets:ViewBudget",
                "budgets:Describe*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AWSBudgetsReadOnlyAccess"
        },
        {
            "Action": [
                "securityhub:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AWSSecurityHubFullAccess1"
        },
        {
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": [
                        "securityhub.amazonaws.com"
                    ]
                }
            },
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AWSSecurityHubFullAccess2"
        },
        {
            "Action": [
                "guardduty:*"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonGuardDutyFullAccess1"
        },
        {
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": [
                        "guardduty.amazonaws.com",
                        "malware-protection.guardduty.amazonaws.com"
                    ]
                }
            },
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonGuardDutyFullAccess2"
        },
        {
```

```
            "Action": [
                "organizations:EnableAWSServiceAccess",
                "organizations:RegisterDelegatedAdministrator",
                "organizations:ListDelegatedAdministrators",
                "organizations:ListAWSServiceAccessForOrganization",
                "organizations:DescribeOrganizationalUnit",
                "organizations:DescribeAccount",
                "organizations:DescribeOrganization"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "AmazonGuardDutyFullAccess3"
        },
        {
            "Action": [
                "iam:GetRole"
            ],
            "Resource":
"arn:aws:iam::*:role/*AWSServiceRoleForAmazonGuardDutyMalwareProtection",
            "Effect": "Allow",
            "Sid": "AmazonGuardDutyFullAccess4"
        },
        {
            "Action": [
                "apigateway:*"
            ],
            "Resource": "arn:aws:apigateway:*::/*",
            "Effect": "Allow",
            "Sid": "AmazonAPIGatewayAdministrator"
        },
        {
            "Action": [
                "sts:GetCallerIdentity"
            ],
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "STS"
        }
    ]
}
```

ANNEX B – ANSIBLE CLIENT

ANSIBLE CLIENT

Ansible is the most popular open source automation tool on GitHub. It is an automation tool used to configure systems, deploy software, and orchestrate more advanced tasks such as continuous deployments or zero downtime rolling updates.

In Maestro solution, Ansible is introduced via the respective wizard on the Main Dashboard.

The wizard allows to generate an Ansible client for the selected region and tenant. The result will be provided as a downloadable zip, containing the dynamic inventory files.

DOWNLOADING ANSIBLE CLIENT

In order to download Ansible Client perform the following steps:

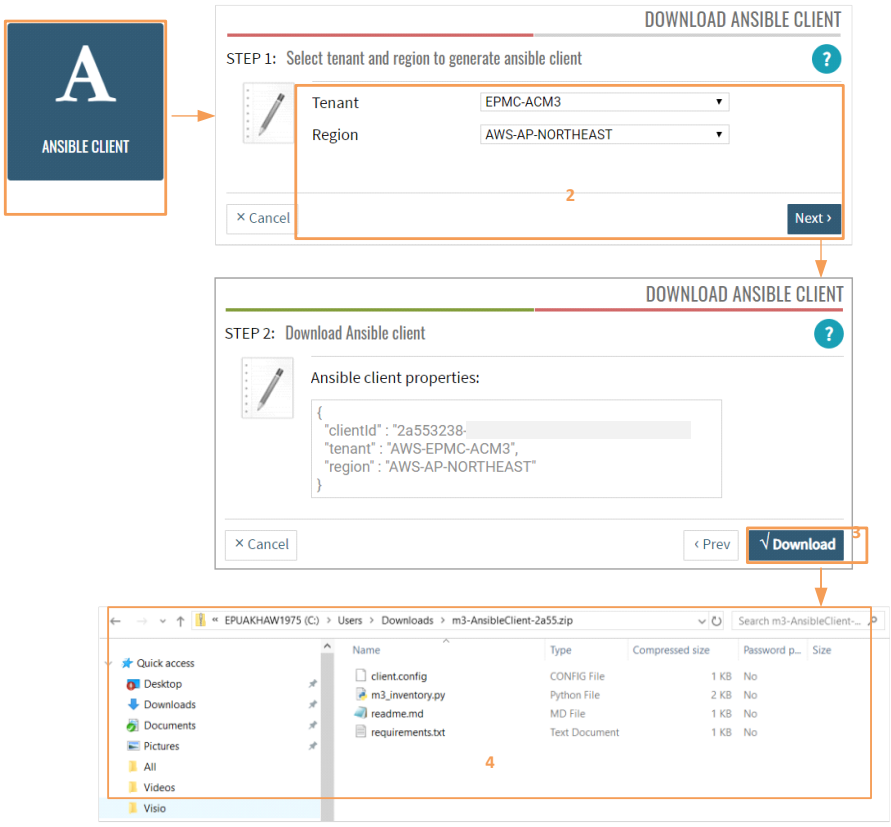


Figure 30 – Ansible Client wizard

1. Start the **Ansible Client** wizard from the main Maestro Dashboard.
2. Specify tenant and region for which you want to generate the Ansible Client parameters and click the **Next** button.
3. In the next window you can review the properties of the file to be downloaded. Click the **Download** button to proceed.
4. The downloaded package will be stored in the **Downloads** folder of your computer.

EXECUTING ANSIBLE CLIENT

To configure Ansible Client in order to get the inventory file, follow the instructions provided in the **readme.md** file from the downloaded package.

The following software should be installed on the machine where the Ansible Client will be executed:

- Python3 v3.7.1 or later
- pip v19.3.1 or later
- virtualenv v16.1.0 or later

In the current revision of Ansible Client (3.2.100.46) only the guide for Linux systems is present in the readme.md file. The flow of work for Linux and Windows is described below in this instruction.

For Linux

1. Navigate to AnsibleClient folder using terminal
2. Create virtualenv using the command
   `virtualenv -p python3 .venv`

Note that your OS may have no configured python3 alias. To configure it properly please follow the instructions by link: https://realpython.com/installing-python/ .

3. Activate virtualenv with the command `source .venv/bin/activate`
4. Install requirements from the requirements.txt file with command
   `pip install -r requirements.txt`
5. Edit client.config file by providing AWS credentials from the environment.
   These credentials should have permissions to invoke lambda function ('lambda:InvokeFunction').

```
#Maestro ansible client config
#Wed Feb 19 10:33:54 UTC 2020
aws_secret_access_key=******************************
clientId=******************************
cache.max.age=300
aws_access_key_id=******************************
lambda.name=autoconf_ansible_meta
lambda.region=eu-central-1
region=AWS-AP-NORTHEAST
tenant=AWS-EPMC-ACM3
```

6. Save the file.
7. Execute the command `python m3_inventory.py`
8. This result will be returned:

```
(.venv) ➜  m3-AnsibleClient-d20c python m3_inventory.py
Ansible DI file has been created by path /Users/user/Downloads/m3-
AnsibleClient-d20c/AWS-EPMC-ACM3.json
```

9. Your Ansible inventory file is ready and is stored in a file located by path returned in output by m3_inventory.py script. In the example this is `/Users/user/Downloads/m3-AnsibleClient-d20c/AWS-EPMC-ACM3.json`.

For Windows

1. Navigate to AnsibleClient folder using cmd.
2. Check the version of the python. Execute ‘python’:

```
(venv) C:\Users\Daryna_kozub\Desktop\m3-AnsibleClient-ffe7>python
Python 3.7.6 (tags/v3.7.6:43364a7ae0, Dec 19 2019, 00:42:30) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3. If the version of python is higher than 3.7.0 everything is ok. If the version is less, install Python3.7+ and use the alias of python3.7 while creating the virtualenv using the command `virtualenv -p python3.7 venv`
4. Create virtualenv with the command `virtualenv venv`
5. Activate virtualenv using the command `.\venv\Scripts\activate.bat`
6. Install requirements using the command `pip install -r requirements.txt`
7. Edit client.config file by providing AWS credentials from the environment. These credentials should have permission to invoke lambda function (‘lambda:InvokeFunction’).
8. Save the file.
9. Execute m3_inventory.py script using the command `python m3_inventory.py`

```
(venv) C:\Users\Daryna_kozub\Desktop\m3-AnsibleClient-ffe7>python
m3_inventory.py
Ansible DI file has been created by path C:\Users\Daryna_kozub\Desktop\m3-
AnsibleClient-ffe7/AWS-EPMC-ACM3.json
```

10. The result has been created and saved to AWS-EPMC-ACM3.json file.

TABLE OF FIGURES

VERSION HISTORY

Version	Date	Summary
2.4	November 12, 2025	Reviewed
2.3	June 5, 2023	Screenshots updated, integration versions updated, permissions for Maestro, Admin Tool updated
2.2	October 20, 2022	Minor details updated, screenshots updated
2.1	March 12, 2021	Minor details updated
2.0	June 17, 2020	Restructured, new sections added, focusing on Mestro3 components
1.4	July 10, 2019	Reviewed, Syndicate concept introduced
1.3	April 18, 2018	Reviewed, several details updated
1.2	January 10, 2018	Updated the Permissions annex
1.1	December 11, 2017	Added the Deployment Models section
1.0	November 04, 2017	Initially published