POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING



MASTER's Degree Thesis

Automated Orchestration and Secure Multi-Tenancy in Hybrid Edge Cloud Environments

Supervisors

Prof. Fulvio RISSO

Ing. Dario SABELLA

Candidate

Loris CATALANO

OCTOBER 2026

Automated Orchestration and Secure Multi-Tenancy in Hybrid Edge Cloud Environments

Loris Catalano

Abstract

The starting point of this work is a private on-premises cloud that sells IaaS capacity, part of it on processors that support TEEs (Trusted Execution Environment). In that baseline system, every step in the life of a tenant was carried out by hand, following written operational procedures. Those procedures take a lot of time and are repetitive and automatable.

This thesis moves that lifecycle from written procedure to executable code in three stages. Terraform modules describe the shape of a tenant, defined as one VDC with its own group, access control rules and quotas. The model was analysed with STRIDE, a framework created by Microsoft to find security risks in software. The third stage puts a natural language interface in front of those operations, driven by an SLM (Small Language Model) served locally, so that no description of the infrastructure leaves the premises. Every request passes a dry run, the policy set, and an operator who approves the plan those two produced.

On the installation the framework was built on, onboarding a tenant takes 57.1 s $\pm$ 6.8 against an estimate of about two hours for the procedure it replaces, and the policy gate costs 0.17 s of it. The audit bundle that offboarding produces takes 7.6 s $\pm$ 0.3. Four models were compared on a corpus of 29 requests and the most accurate answered 28 correctly. No model ever selected the wrong operation on a request that names one.

Everything was built and measured on a laboratory cluster nested inside a single physical host, which reproduces the structure of the production installation but not its behaviour, so these figures cannot be read as production figures.

Table of Contents

1 Introduction **1**

1.1 Context . . . 1
1.2 Goal of the Thesis . . . 2
1.3 Thesis Structure . . . 3

2 Problem Statement and State of the Art **5**

2.1 The Baseline Tenant Lifecycle . . . 5
2.2 Problem Statement . . . 6
2.3 Infrastructure Automation . . . 7
2.4 Tenant Provisioning in Existing Clouds . . . 8
2.4.1 Public Cloud Providers . . . 8
2.4.2 On-premise and Private Cloud . . . 9
2.5 Policy as Code . . . 10
2.6 Natural Language Interfaces to Infrastructure . . . 12
2.6.1 Generating Infrastructure Code . . . 12
2.6.2 Conversational Operation of Existing Tools . . . 13
2.7 Discussion . . . 14

3 Technologies Used **16**

3.1 Ansible . . . 16
3.2 Terraform . . . 18
3.2.1 The declarative model and the resource graph . . . 18
3.2.2 Providers . . . 19
3.2.3 State . . . 19
3.2.4 The plan and apply cycle . . . 20
3.2.5 Modules and state roots . . . 21
3.2.6 Configuration management and provisioning . . . 21
3.3 OpenNebula . . . 22
3.3.1 Front-end, nodes and drivers . . . 22
3.3.2 The resource model . . . 23
3.3.3 Users, groups and the primitives of confinement . . . 25
3.4 Natural Language Interface Stack . . . 26
3.4.1 Serving a model locally . . . 26
3.4.2 Tool calling . . . 27

4 System Architecture **29**
4.1 Overview . . . 29
4.2 Architectural Layers . . . 31
4.2.1 The vertical stack . . . 31
4.2.2 Partitioning the declaration . . . 32
4.3 Components and Responsibilities . . . 34
4.3.1 The interface . . . 35
4.3.2 The agent . . . 35
4.3.3 The operations . . . 35
4.3.4 The judges and the record . . . 36
4.4 Interaction Flows . . . 37
4.4.1 Onboarding a tenant . . . 37
4.4.2 Offboarding a tenant . . . 38
4.4.3 Changes to a running tenant, and standalone machines . . . 39
4.4.4 When a check refuses . . . 40
4.5 Integration with OpenNebula . . . 41
4.5.1 What the platform decides . . . 41
4.5.2 What the framework declares . . . 41
4.5.3 Where the deployment stops . . . 42

5 Security Model, Measures and Limitations **44**
5.1 Tenant Model . . . 44
5.1.1 What a tenant is . . . 44
5.1.2 The invariants of the model . . . 46
5.2 Separation and Isolation Mechanisms . . . 47
5.2.1 Ownership and authorisation . . . 47
5.2.2 Network separation . . . 48
5.2.3 Compute and placement . . . 49
5.2.4 Storage . . . 49
5.2.5 Where a cross-tenant call stops . . . 50
5.3 Threat Analysis . . . 50
5.3.1 Adversaries and what is trusted . . . 51
5.3.2 Hypervisor and authorisation layer . . . 52
5.3.3 Network layer . . . 54
5.3.4 Storage layer . . . 56
5.3.5 Residual risks . . . 57
5.4 Security Guarantees . . . 57
5.5 Limitations . . . 58

6 Implementation **60**
6.1 Integrating Terraform and OpenNebula . . . 60
6.1.1 The provider and its credentials . . . 60
6.1.2 From tenant primitives to resource types . . . 60

6.2 Ansible Automation . . . 61
6.2.1 The lab and the deployment chain . . . 61
6.2.2 Additions to the collection . . . 63
6.3 Code and Module Organization . . . 64
6.3.1 Languages and repositories . . . 64
6.3.2 Modules and their interfaces . . . 65
6.3.3 Provider and packaging limitations . . . 66
6.3.4 The operations as shell scripts . . . 67
6.3.5 Policies and continuous integration . . . 67
6.4 Natural Language Interface Implementation . . . 68
6.4.1 Three processes on one machine . . . 68
6.4.2 The tool layer . . . 69
6.4.3 Making the output of a small model usable . . . 69
6.4.4 The backend, the plan and the approval . . . 70

7 Evaluation and Results 72
7.1 Test Methodology and Experimental Setup . . . 72
7.1.1 The measurement environment . . . 72
7.1.2 What the metrics mean . . . 73
7.2 Tenant Lifecycle: Automated vs. Manual Process . . . 73
7.2.1 The cost of one lifecycle . . . 74
7.2.2 Destruction costs four times creation . . . 75
7.2.3 Comparison with the manual process . . . 75
7.3 Comparison of different models . . . 76
7.3.1 The corpus and the candidates . . . 76
7.3.2 Accuracy, arguments and latency . . . 77
7.3.3 Three ways of failing . . . 77
7.4 Discussion . . . 78

8 Conclusions and Future Work 80
8.1 Summary of the Work . . . 80
8.2 Assessment of the Approach . . . 81
8.2.1 The infrastructure layer . . . 81
8.2.2 Checking before applying . . . 81
8.2.3 The natural language layer . . . 82
8.3 Future Work . . . 82
8.3.1 Short-term Extensions . . . 82
8.3.2 Long-term Extensions . . . 83
8.3.3 Towards Production Deployment . . . 84

Bibliography 85

List of Figures

3.1 The path of one task, from the control node to the managed nodes. . 17
3.2 The three terms Terraform works with, and the two commands that relate them. 20
3.3 Where each tool meets the platform. 24
3.4 The two shapes a reply can take. 28

4.1 The boundary of the framework, and what is left outside it. 30
4.2 The five layers, what each accepts, and what it hands down. The labels on the left name the checks a layer runs on what it receives. . 32
4.3 The four state roots, their inputs, and the dependencies among them. The dashed line separates the ordered chain from the standalone root. 34
4.4 One onboarding request, from the sentence to the applies. 38
4.5 The four points at which a change can be refused, and what each refusal leaves behind. 40

5.1 The ten kinds of object a tenant is made of, and what stays outside every tenant. 45
5.2 Four cross-tenant calls, the mechanism that refuses each of them, and the object each one never reaches. 50
5.3 Where each adversary sits, what the analysis treats as trusted, and what it leaves outside. 52

6.1 The three levels of the lab installation. The compute node is itself a guest of the machine that runs the front-end. 62
6.2 The three processes of the interface, and what the tool layer reaches below them. 69

7.1 Where the time of one lifecycle goes. The removal bar is the decomposition of the offboarding ceremony, measured over three repetitions. . 74
7.2 The five repetitions. Each dot is one run, the vertical rule is the mean and the band is one standard deviation on either side of it. 75
7.3 How the 29 scenarios of one repetition end, for each model. 78

List of Tables

2.1 Mechanisms available for issuing and confining a tenant. 10

3.1 Ansible and Terraform along the axes that separate configuration management from provisioning. 22

4.1 The six operations, with the databases each one edits and the roots it applies. 36
4.2 Who creates each object the system depends on. 42

5.1 Each element of the model, the primitive that realises it, and the module that creates it. 46
5.2 Each invariant, the mechanism that sustains it, and what happens to a call that would break it. 51
5.3 Threats at the hypervisor and authorisation layer. 53
5.4 Threats at the network layer. 55
5.5 Threats at the storage layer. 56

6.1 The resource types that realise a tenant, and the module each one is declared in. 61
6.2 The languages of the three repositories, and the size of each part. . . 64

7.1 Software the measurements ran against. 72
7.2 The two measured campaigns behind this chapter. 73
7.3 Duration in seconds of each step, over five complete cycles. 74
7.4 The manual procedure and the automated pipeline. 75
7.5 The candidate models, largest first. 76
7.6 Four models over 29 scenarios. Latency is per request, over 87 requests for the models with three repetitions. The argument score of mistral-7b is computed over its three abstentions, which have no arguments to fill. 77
7.7 Scenarios answered correctly by category, for one repetition. The three repetitions coincide. 77

Acronyms

IaC	Infrastructure as Code.
IaaS	Infrastructure as a Service.
TEE	Trusted Execution Environment.
VM	Virtual Machine.
API	Application Programming Interface.
CLI	Command Line Interface.
STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege.
VDC	Virtual Data Center.
VXLAN	Virtual Extensible LAN.
OPA	Open Policy Agent.
SLM	Small Language Model.
LLM	Large Language Model.
SCP	Service Control Policy.
VPC	Virtual Private Cloud.
SSH	Secure Shell.
HCL	HashiCorp Configuration Language.
XML-RPC	XML Remote Procedure Call.

GPU	Graphics Processing Unit.
VNI	VXLAN Network Identifier.
SGX	Software Guard Extensions.
TDX	Trust Domain Extensions.
EPC	Enclave Page Cache.
VPN	Virtual Private Network.
PCI	Peripheral Component Interconnect.
EVPN	Ethernet Virtual Private Network.

Chapter 1

Introduction

1.1 Context

The scope of this thesis is a private cloud built on OpenNebula, on which a provider sells IaaS capacity to its customers. Part of the compute fleet runs on processors that support TEEs, so that workloads handling sensitive data can be confined to a hardware-isolated enclave.

This thesis describes two different roles that act on that platform. A **tenant** is one customer together with the resources allocated to it: user accounts, virtual networks, disk images, quotas, and running VMs. A tenant consumes those resources through its own account on the platform, sees no resource belonging to another tenant, and never runs the automation that created them. An **operator** is a member of the cloud provider's staff: operators hold administrative credentials on the OpenNebula front-end, run the automation described in this thesis, and are accountable for the changes it applies. The two roles meet in the same organisation only for a customer that takes the whole framework in house and runs its own installation.

The starting point of this work is a baseline system, where an OpenNebula platform is deployed and every step of a tenant's life is performed by hand. A set of internal operational procedures prescribes what to do and in which order: how a customer requests the service, how a tenant is provisioned on encrypted storage, how its VMs are backed up, how a host is hardened, how the encryption keys are rotated. Each document lists an ordered sequence of actions on the OpenNebula front-end that an operator carries out through the web interface or the command line. The procedures are detailed and the operators follow them, but the only thing that guarantees a tenant is configured as intended is the care taken while executing them. Three consequences follow:

- Repeatability rests on the operator: two operators, or the same operator on two different days, can produce two configurations that both satisfy the written procedure and still differ in a quota, a permission bit, or a network attachment.
- Security-relevant constraints, such as which images may be instantiated, where a tenant's data may reside, and how much of the cluster it may consume, are

written in prose and checked by reading. As a result, a violation surfaces only when someone notices it, or when a tenant reaches a resource it should not see.

- Not every resource records why it exists: deciding whether a virtual network still belongs to an active customer can mean inspecting the running state and reconstructing the intent behind it.

The result is a platform whose configuration cannot be reproduced, audited, or verified except by inspecting it. This thesis closes that gap by moving the tenant lifecycle from written procedure to executable code: the resources of a tenant are declared in version-controlled Terraform, the security constraints are expressed as machine-checked policies evaluated before any change is applied, and an operator can drive the whole process by describing the intended outcome in plain language. Chapter 2 examines the existing procedures in detail and states the problem they pose.

1.2 Goal of the Thesis

For this work, closing the gap means making a tenant operation explicit before it touches the cluster. A new tenant, a quota change, or an offboarding should be described as version-controlled infrastructure code. The intended change should be checked against the same isolation and placement rules every time, and the final apply should remain a deliberate operator action. The goal is therefore not to replace operators, but to move the fragile part of the process, remembering each manual step and verifying its side effects, into artifacts that can be reviewed, tested, and repeated.

The implementation follows that goal in three stages. The first expresses the OpenNebula resources that make up a tenant, from identities and networks to storage and compute, as Terraform modules that can be applied, changed, and destroyed without manual steps. The second defines what it means for tenants to be separated in that model and encodes the definition as policy, so that an invalid plan is rejected before it reaches the OpenNebula front-end. The third adds a local natural language interface on top of those operations, so that a request such as onboarding a new customer can be written as a sentence and still end in a Terraform run that an operator has reviewed and approved.

This thesis makes four contributions:

1. An IaC framework for OpenNebula 7.0, built on the official Terraform provider and organised as four independent state roots: the shared platform, the tenant definitions, the tenant workloads, and standalone virtual machines. We kept the state separate so that a change to one tenant cannot plan a change to the platform underneath it. Continuous integration checks formatting, policy compliance, and validity of every root on each change.

2. A tenant model expressed in OpenNebula primitives: one VDC per tenant, with its own group, access control rules, and quotas, combined with network micro-segmentation that gives each tenant a VXLAN-backed virtual network reachable only through a virtual router. The model was analysed with STRIDE, and the lifecycle is automated from onboarding to an offboarding that exports an audit bundle before destroying anything.

3. A policy layer that evaluates every planned change before it is applied, covering data residency, image provenance, and per-tenant quotas, and placement of workloads on nodes equipped with a TEE, first exercised against simulated node labels and later on real hardware.

4. A natural language interface to the operations above, driven by a SLM served locally rather than by a cloud API, so that no description of the infrastructure leaves the premises. Every request passes through a dry run, the policy check, and an explicit human approval before any change is applied. Different models were compared on a corpus of 29 provisioning scenarios, the best of which selected the correct operation in 28 cases out of 29 (Chapter 7).

The natural language interface is the original contribution; the first three stages are the engineering work that makes it safe to expose infrastructure operations that way.

Everything described here was built and validated in a test environment: a single physical host running a nested virtualization stack, with the OpenNebula front-end and its compute nodes as virtual machines and the tenant workloads nested inside them. That environment reproduces the structure of the production cluster, not its behaviour and contents, therefore results measured on it cannot be read as production figures. Porting the framework to production was under way while this thesis was being written, and Chapter 8 describes what it requires.

Part of what the following chapters describe belongs to a running commercial installation, so some of it appears here under a pseudonym. Addresses, subnets and host names are replaced by fictitious values, and customers are named `tenant-a`, `tenant-b` and so on. The substitutions are fixed once and used consistently across text, figures and listings. Software versions, hardware sizing, module and file names, and every measured time are the real ones.

1.3 Thesis Structure

The seven chapters that follow divide into the background the work starts from, the work itself, and its measurement:

- Chapter 2 states the problem the manual lifecycle poses, and reviews what account factories, private cloud stacks, policy engines and conversational assistants already do about it.

- Chapter 3 describes Ansible, Terraform and OpenNebula, and the components that serve a language model locally.
- Chapter 4 defines the five layers a request crosses before it reaches a host, the four state roots the declaration is split into, and the six operations that edit them.
- Chapter 5 defines a tenant as ten kinds of object, names the mechanism that separates one tenant from another, and reports a security analysis together with the risks it leaves standing.
- Chapter 6 covers the Terraform modules and their interfaces, the deployment of the laboratory cluster, the policy set and the pipeline that evaluates it, and the three processes behind the natural language interface.
- Chapter 7 times one complete tenant lifecycle against the manual procedure it replaces, and compares four models over a corpus of 29 requests.
- Chapter 8 assesses what each of the three stages achieved and what it did not, and states what porting the framework to production requires.

Chapter 2

Problem Statement and State of the Art

2.1 The Baseline Tenant Lifecycle

In the baseline system, the life of a tenant is governed by internal documents rather than by a tool. They cover the customer's request for the service, the activation of the resources, the recurring operations on a live tenant (encryption of its storage, rotation of the encryption keys, backup of its virtual machines, hardening of the hosts underneath) and the closing of the contract. Each document is prose, each prescribes an ordered sequence of actions, and each is carried out by hand on the OpenNebula front-end.

A customer buys capacity on the private cloud: virtual machines of a given size, storage that only it can read, a network segment of its own, and a quota that bounds how much of the cluster it may consume. The customer administers the machines it runs, while the cloud provider that manages OpenNebula administers everything underneath them from the hypervisors to the storage on which the tenant's disks live.

Activation follows a written process, approved internally and aligned with the information security controls the cloud provider is certified against. It runs from the contractual formalities and the collection of the customer's requirements through a risk assessment, the design of the tenant architecture, the technical activation of the resources, a security verification before the tenant is allowed into production, and a formal approval. The same process governs the reverse path: at the end of the contract the tenant is decommissioned, its data securely erased, and a certificate of deletion issued.

Technical activation is a procedure of its own, composed of ordered phases carried out on the OpenNebula front-end. Capacity is sized and checked, the tenant's encrypted volumes are created, those volumes are registered as datastores, access control rules and cluster placement are configured, base images and machine templates are prepared, the machines are instantiated and their backup configured and tested, the encryption keys are placed in custody, and ownership of everything is transferred

to the tenant. Every phase is a sequence of commands typed in a fixed order, and a single tenant takes several tens of them.

The procedure assigns the technical phases to one role, the technical lead, and the clearance for production to another, the security manager. Alongside the running infrastructure, each activation produces paperwork: an entry in the activation register, an entry in the key register, a hardening checklist, and a pre-production verification report. All of it is compiled by hand, after the fact, from what the operator did.

2.2 Problem Statement

The procedure is precise and the operators follow it, but the difficulty is that it is executed rather than declared, and four consequences follow from that:

- Every activation is serialised on one person. The manual work takes about one hour for a single VM and about two hours for a full tenant, but the tenant becomes usable only when the technical lead is free to run the procedure and the security manager is free to sign off, so the elapsed time is set by availability rather than by the work.
- The written procedure fixes the order of the actions, not the result. Two executions that both respect it can still differ in a quota, in a permission bit, or in which cluster a datastore was attached to, and nothing afterwards compares the tenant that exists against the tenant that was designed. Detecting that difference means reading the live configuration and reconstructing what it should have been.
- The constraints that matter for security are written in prose. A tenant may only instantiate images from an approved catalogue, its data must stay on storage inside a given boundary, and it may not exceed the capacity it paid for. Each of these is a sentence in a document, checked by the person executing the procedure. A machine can enforce none of them, so a violation is caught by review or not at all.
- The records that make the process auditable are produced by hand, and they cost more time than the commands they describe. They are also derived data: which datastore was created, under which identifier, with which quota and which owner is information the platform already holds. Compiling it manually into a register duplicates what the system knows, and does so in a form that can drift from it.

Not all of this can, or should, be automated. What this thesis addresses is the stretch that runs from registering the tenant's storage to handing it ownership of its resources, together with the reverse path at decommissioning.

Automating that stretch imposes a set of requirements on the solution:

1. A tenant should be declared once and applied as many times as needed, with the same declaration producing the same result.
2. The constraints on a tenant should be evaluated by a machine before a change reaches the cluster, not read afterwards.
3. The evidence that the process was followed should fall out of the process itself, rather than being written up separately.
4. None of this may depend on a service outside the premises, since the platform exists to keep customer data under the control of the cloud provider.

2.3 Infrastructure Automation

What separates a written procedure from an executable one is that a declaration can be compared with reality and a procedure cannot. Infrastructure as Code (**IaC**) is the practice of writing the configuration and provisioning of machines, networks and storage as source code, kept under version control and executed by a tool that reconciles the running system with the state the code describes [1]. A script fixes the commands and their order, while a declaration fixes the resources that should exist and leaves the tool to compute what to do about the difference. The procedures of Section 2.1 belong to the first category, written in prose rather than in a language, which is why two executions that both follow them can leave two different tenants behind.

The practice grew out of **DevOps**, defined as the organizational effort to automate continuous delivery while keeping releases correct and reliable [2]. IaC is the part of that effort that applies to the machines rather than to the software running on them. **GitOps** takes the idea one step further: the version-controlled declaration becomes the only accepted source of truth, and an agent continuously reconciles the running system against the repository instead of waiting for an operator to trigger a deployment [3].

Serverless computing moves in the opposite direction. The provider allocates and scales the resources on demand and bills for execution, so the customer writes no infrastructure description at all [4]. That model does not remove the provisioning work, it relocates it: someone still owns the machines the functions run on, and on a private cloud that someone is whoever operates the platform. Serverless therefore changes what the customer has to describe and leaves untouched the layer this thesis works on, where machines, networks, storage and quotas still have to be created, isolated and accounted for.

Whether the practice delivers what it promises has been studied directly. Interviews with senior developers at 44 companies found IaC in wide use for provisioning, with Terraform the most common tool, and a consistent complaint about what the tooling does not cover: testing and maintenance [5]. The shape of that gap is worth noting, because it recurs in this thesis. The available tools are good at creating

infrastructure and weak at telling the operator whether the description was right in the first place.

Infrastructure code carries defects like any other code, and they can be measured. Seven categories of security smell were derived from a qualitative analysis of 1,726 scripts, and a linter built for the purpose then found them in 15,232 scripts collected from open source repositories [6]. Defective scripts can also be predicted within a project from product and process metrics, a result obtained across 104 repositories [7]. Applying ordinary software quality practice to infrastructure code is thus well supported, and it is a necessary condition for what follows. It is not a sufficient one: a description can pass every linter, apply without error, and still produce a tenant that violates a constraint nobody wrote down in a checkable form.

One defect class is specific to the paradigm. State reconciliation is the step in which a tool queries the current state and compares it with the desired one, and an analysis of 5,110 such defects produced eight categories, three of which had not been reported for other kinds of software [8]. The most frequent concerns the inventory, the tool's own record of what exists. The failure mode is uncomfortable because the code can be correct while the record of reality is wrong, and the tool will act on the record. How a framework partitions that record is therefore a design decision rather than an implementation detail.

2.4 Tenant Provisioning in Existing Clouds

The problem of handing a customer an isolated slice of a shared platform is not new, and the public providers like Amazon, Microsoft and Google have solved a version of it at a scale that is different from the scope of the present thesis work. Their answers converge on a recognisable shape, which is worth extracting before looking at what the private cloud stacks offer.

2.4.1 Public Cloud Providers

There are four mechanisms that recur across the three major providers:

- The tenant boundary is a container issued by the platform, not a set of permissions arranged inside a shared container.
- Those containers are placed in a hierarchy.
- Policies attach to nodes of the hierarchy and are evaluated by the platform when a request is made, so an administrator inside the container cannot lift them.
- A provisioning service issues a new container from a template, which is what turns onboarding from a procedure into an operation.

On **AWS** (Amazon Web Services) the container is an account. Organizations groups accounts into organizational units, and SCPs attached to an organizational

unit bound what the identities in the accounts below may do. Control Tower assembles a landing zone on top of Organizations, Service Catalog and IAM Identity Center, and its Account Factory issues new accounts from a configurable template [9]. Its controls come in three kinds:

- preventive, which stop an action;
- detective, which report a non-conforming resource;
- proactive, which check a resource before it is provisioned.

Control Tower also detects the divergence of an account from the configuration the landing zone established for it, called drift.

On **Microsoft Azure** the container is a subscription, management groups form the hierarchy, and Azure Policy assignments attach to management groups. Microsoft documents the process of issuing a subscription as subscription vending [10]. An intake collects the anticipated budget, the owner, the networking requirements, and a classification of business criticality and confidentiality. A central team then approves the request, and a pipeline creates and configures the subscription from published Bicep or Terraform modules. The classification collected at intake determines which management group the subscription lands in, and therefore which policies apply to it. Two details matter for this thesis: the approval is part of the design, and the automation is expected to be IaC driven by a pipeline rather than a bespoke service.

Google Cloud organizes resources hierarchically into an organization containing folders and projects. The container is the project, which hosts a workload and the resources it uses, and constraints from the Organization Policy Service can be enforced at any level of the hierarchy [11]. Network separation is expressed through shared VPC networks and through VPC Service Controls perimeters, which bound where data may move. Google does not sell a managed account factory: its landing zone guidance recommends applying IaC principles [12], and the concrete implementation is a set of Google-maintained Terraform modules (the **Cloud Foundation Toolkit**) whose project factory module creates projects from a declaration [13]. Even on a platform that owns every layer underneath, the answer to tenant provisioning is a set of modules rather than a service, which is the same construction this thesis adopts on OpenNebula.

2.4.2 On-premise and Private Cloud

The three systems below are open source platforms that manage compute, storage and network on hardware their operator owns, and each defines a tenant in its own vocabulary. OpenStack and CloudStack are general purpose alternatives to a public cloud; OpenNebula is the platform considered in this work, and Chapter 3 describes it in detail.

OpenStack models identity in Keystone: a domain is an administrative boundary containing projects, users and groups. A project owns the resources and carries the quotas, and roles are granted to a user or to a group on a project [14]. The primitives

describe a tenant completely, and what Keystone does not have is the workflow that issues one. Adjutant exists to fill that gap: a pluggable workflow service whose stated purpose is to supply functionality missing from Keystone, and whose tasks are built around three states: initial submission, administrator approval, and token submission by the requesting user [15]. An approval gate appears here for the same reason it appears in Azure's vending process, and it is a deliberate part of the model rather than an obstacle to automation.

CloudStack draws the boundary at the account, which represents a customer or a department and may contain several users. Resources belong to the account rather than to the user who created them, and billing and resource limits are maintained per account; accounts are grouped into domains whose administrators see their own subtree and not the physical infrastructure [16]. Users inside one account are not isolated from each other but are isolated from other accounts, and stronger separation is obtained by dedicating zones, pods, clusters or hosts to an account or a domain.

OpenNebula offers the same kind of vocabulary: users and groups, VDCs that scope which hosts, datastores and virtual networks a group can reach, permissions and access control rules, quotas, and per-role interface views [17]. Chapter 3 describes these primitives more in detail and Chapter 5 builds the tenant model on them. Table 2.1 shows that the three private cloud stacks provide the objects a tenant is made of and stop there. There is no hierarchy above the boundary, no policy engine that the platform consults before accepting a request, and no service that issues a tenant from a template. Those three absences are the reason the procedures of Section 2.1 exist in the form they do, and they are what an automation framework for such a platform has to supply itself.

Table 2.1: Mechanisms available for issuing and confining a tenant.

Platform	Tenant boundary	Grouping and platform-evaluated policy	Provisioning service
AWS	Account	Organizational units; SCPs; preventive, detective and proactive controls	Control Tower Account Factory
Azure	Subscription	Management groups; Azure Policy assignments	Subscription vending pipeline
Google Cloud	Project	Folders; Organization Policy Service constraints	Cloud Foundation Toolkit modules, no managed service
OpenStack	Project	Domains; no policy engine	Adjutant, external to Keystone
CloudStack	Account	Domains; no policy engine	None
OpenNebula	VDC and group	Flat, no hierarchy; no policy engine	None

2.5 Policy as Code

Expressing a constraint as code means writing it in a language a machine evaluates, and keeping that evaluation separate from the component that acts on the answer. A constraint written in a document is checked by whoever reads it. The NIST guide

to attribute-based access control describes an access control mechanism split into four functional points [18]:

- a policy decision point that renders the decision;
- a policy enforcement point that applies the decision;
- a policy information point that supplies the attributes the decision needs;
- a policy administration point where the rules are managed.

Policy as code applies that split to configuration rather than to access requests: what is judged is a proposed change to the infrastructure.

OPA (Open Policy Agent) is a graduated project of the Cloud Native Computing Foundation, and it implements the split. It is the engine this thesis uses. The caller submits a structured input document (usually JSON), the engine evaluates rules written in Rego against that input together with any data loaded into it, and it returns a decision that may be a boolean or a structured value [19]. Nothing in the engine knows what the input represents, which is why the same tool serves API authorization, container admission and infrastructure plans.

Where the decision point sits in the pipeline decides what it is worth. In Kubernetes an admission controller intercepts a request after authentication and authorisation and before the object is persisted, and a validating admission controller can reject the request without modifying it [20]. The equivalent position for infrastructure code is between the plan and the apply. HCP Terraform occupies it: for every run it checks the plan against a policy set written in Sentinel, in Rego or in HCL, and stops the run according to the enforcement level of the failing policy [21]. Three levels exist:

1. Advisory: reports the failure and lets the run continue.
2. Soft mandatory: stops the run but can be overridden by an authorised user.
3. Hard mandatory: stops the run until the cause is fixed.

A check placed after the change is a report, and one placed before it is a gate. The graduated levels are an admission that the distinction is not binary in practice: a policy strict enough to be worth writing is often strict enough to block work that ought to proceed, and someone has to decide who carries that exception. Policies are code, so they can be wrong, and the providers treat them as objects to be verified rather than trusted. AWS has built two systems on that premise, one that analyses the policy language it already had and one that replaces it. **Zelkova** encodes the semantics of an AWS policy into logic a solver can decide, which allows questions such as whether one policy is strictly more permissive than another to be answered rather than argued, and it is reported to run millions of times a day [22]. **Cedar** was designed from the start so that a policy language can be fast to evaluate and still amenable to automated analysis, with the language modelled in a proof assistant

and its properties proved [23]. Rego has no comparable decision procedure. What its toolchain offers instead is a unit-test runner: rules prefixed with `test_` are executed by `opa test`, which also reports coverage of the policies under test [24]. The gap between proving a property and testing a sample of cases is real, and it is the reason the policy set in this work is small and each rule is covered by tests rather than inspected by eye.

Two questions are left open by the tools themselves. The first is where the input document comes from, since a policy is only as good as the description of the change it is given to judge. The second is who may override a decision, and under what record. Both are answered by the system that embeds the engine, not by the engine, and Chapter 4 and Chapter 5 answer them for the framework built in this thesis.

2.6 Natural Language Interfaces to Infrastructure

In natural language interfaces applied to infrastructures, a request typed as a sentence (e.g. *onboard a new customer with two machines*) can be answered in two ways. A system can write the infrastructure code that would produce the result, or it can match the sentence against operations that already exist and fill in their parameters. The choice decides what can be wrong in the output, and therefore what has to be checked before the change reaches the cluster. Most of the published work takes the first route.

2.6.1 Generating Infrastructure Code

A survey of the area collects the models, the target platforms and the open problems of producing IaC from a description in natural language [25]. Evaluation benchmarks are used to measure the performance of the models. IaC-Eval is a benchmark of 458 hand-curated scenarios over AWS services, each pairing a description with a specification of the infrastructure it intends; GPT-4 reaches a pass@1 of 19.36% on it, the fraction of scenarios solved on the first attempt, against 86.6% on EvalPlus, a benchmark of ordinary Python problems [26]. The distance between the two figures is the useful part: what a model scores on ordinary code says little about what it will do with a Terraform configuration.

A second benchmark shifts the criterion from syntactic correctness to deployability. DPIaC-Eval collects 153 scenarios across 58 AWS services, and six current LLMs deploy between 20.8% and 30.2% of them at the first attempt; IaCGen, a loop that feeds format, syntax and deployment errors back to the model, raises that figure to between 54.6% and 91.6% within ten iterations [27]. What those templates contain was measured separately. Among the ones produced at the last iteration, 25.2% carry both the resources and the attributes the request specified, and among the ones that deploy, a Checkov scan finds 8.4% fully compliant against a pass rate of 75.3% on the individual policies: a failure on one rule invalidates the whole template. A configuration that applies without error is not a configuration that does what was

asked, which is the argument for showing the operator the change before it is applied and for judging it against rules written independently of the model.

The closest published system by construction is ARPaCCino, an agentic pipeline with retrieval augmentation that generates Rego rules from a description in natural language, evaluates Terraform configurations against them, and iterates corrections until the configuration complies. The authors report that it works with small open-weight models [28]. It shares the local model, Rego and Terraform with the interface built in this thesis, and it reverses the direction of the trust. There, it is the model that writes the policy; here the policy set is fixed and is what judges what the model proposes. A generated rule inherits the uncertainty of the generation, which is tolerable when the output is a compliance report and not when it is the gate in front of a `terraform apply`.

2.6.2 Conversational Operation of Existing Tools

A second type of design puts the conversation in front of operations written in advance. Storey and Zagalsky describe bots as an interface between developers and the tools and services of their work, one that mediates access to them rather than producing new artifacts, and ask what that shift does to productivity [29]. This is the family the interface of this thesis belongs to: the model selects among a closed set of operations and supplies their arguments, while the operations themselves are ordinary code, reviewed once and executed unchanged.

Practice has moved ahead of the literature, and each of the three public providers now ships a conversational assistant over its own control plane.

- **Amazon Q Developer** answers questions about an account and its resources from the AWS Management Console and from an integrated development environment, and runs on Amazon Bedrock, the provider's managed model service [30].
- **Azure Copilot** works in the Azure portal, reaches only the resources the signed-in user may already reach, and requires that user's confirmation for any action it takes [31].
- **Gemini for Google Cloud** generates and refines Google Cloud CLI commands, `kubectl` manifests and Terraform code through a conversation in the Application Design Center [32].

Two properties are common to the three: the model runs as a managed service inside the provider's cloud, which is what requirement 4 of Section 2.2 excludes, and the confirmation before acting is written into the design rather than bolted on afterwards, which is the position of the approval step in this work as well.

There are two absences that are worth recording explicitly. The first is that ChatOps is common in industry and its academic record is thin, the paper cited above being the most solid reference available. The second is that no peer-reviewed work we could find places the human approval inside the pipeline, after a machine-evaluated

policy decision and before the change is applied. That combination is what Chapter 4 builds and Chapter 7 measures.

2.7 Discussion

Each of the four requirements of Section 2.2 is met in this chapter. The public providers solve tenant provisioning inside a platform they own end to end. The account, subscription or project is issued by the control plane, the hierarchy above it is a structure the control plane maintains, and the policy attached to that hierarchy is evaluated by the control plane before a request is served. None of the three is a component that can be installed on a cluster the provider does not operate. What is considered in this thesis is the shape summarised in Table 2.1, not the mechanism.

As stated before, the private stacks supply the objects a tenant is made of and stop there. A VDC, a group, a set of access control rules and a quota describe a tenant in OpenNebula, but nothing issues one from a declaration or refuses a configuration that breaks a rule stated elsewhere. OpenStack is the only one of the three with a workflow that issues a tenant, and it lives in Adjutant, a service separate from Keystone. Requirements 1 and 2 therefore have to be satisfied one layer above the platform, by tools that know nothing about tenants until they are told.

Those tools are available and their suitability is not in question. Terraform declares resources and reconciles them, OPA evaluates a structured description of a proposed change, and the split between plan and apply gives the decision point somewhere to stand, as HCP Terraform shows commercially. What none of them contains is the mapping from a tenant onto the OpenNebula primitives that implement it, nor the rules that decide when an instance of that mapping is valid. Writing that mapping down, and choosing which of its properties are worth checking before an apply rather than after, is the work of Chapter 4 and Chapter 5.

Requirement 3 has the least support of the four. OpenNebula records accounting and showback, which say what a tenant consumed and not why a resource exists or who cleared it for production. The nearest precedent is drift detection in AWS Control Tower. Without an equivalent on the platform, the audit material has to be a product of the automation. The declaration already states what a tenant should be, and a run already knows what it changed, so the register that Section 2.1 describes as hand-compiled is a serialisation of two artifacts the framework already holds.

The natural language layer inherits a different mismatch. The benchmarks of Section 2.6.1 score a system asked to author something new for every request. The lifecycle of Section 2.1 does not need it: onboarding a tenant, scaling it, changing its quota and decommissioning it are four operations, each parameterised by a handful of values. Reducing the model's task from writing a configuration to choosing an operation and filling its arguments removes most of the surface those benchmarks measure, and leaves a smaller one that Chapter 7 measures instead.

Requirement 4 settles what is left. The three provider assistants are managed services, so a description of our infrastructure would have to leave the premises to

reach them, and the model therefore has to be small enough to be served on available hardware. That bounds the accuracy the interface can offer, and this thesis reports it.

Chapter 3

Technologies Used

The work in Chapters 4 to 7 is built on three technologies. They are the industry standards for configuration management, infrastructure as code, and on-premises cloud platform management. **Ansible** brings a set of machines to the state in which they run a cloud platform, **Terraform** declares the resources that platform has to expose, and **OpenNebula** is that platform and the API at which both of the others stop. A separate stack of components serves the natural language interface, and the last section describes it. Each section below gives the model of operation and the vocabulary used later, and refers to the official documentation. How these technologies are combined is described in Chapter 4 and Chapter 6.

3.1 Ansible

Ansible is an open source tool for the automation of system administration: it configures the software installed on a machine, deploys applications onto it, and runs sequences of operations across several machines at once. It is written in Python, distributed under the GNU General Public License version 3 or later, and sponsored by Red Hat [33]. What ships is `ansible-core`, the engine together with a base set of modules, extended by collections that package further modules, roles and plugins. The work described in this thesis uses `ansible-core` 2.20.5.

Operations run from a single control node, the machine where the command line tools are invoked, against a set of managed nodes, the devices a run is aimed at [34]. The managed nodes carry no permanently installed component of the tool: the control node opens a Secure Shell (SSH) connection with credentials the target operating system already accepts. Avoiding the installation of further software across the infrastructure is a stated design goal of the project [35]. Figure 3.1 shows the path a single task travels.

A module is the code or binary that Ansible copies to and executes on each managed node to carry out the action a task names. That mechanism sets the requirements on the target rather than on the network. Most modules are Python programs, so an interpreter has to be present, and the tasks that cannot assume one fall back on `raw`, which sends a command straight to the shell and bypasses the

module subsystem entirely. The module writes a JSON document to its standard output, and the temporary copy is deleted when the task ends unless the operator asks for it to be kept for debugging.

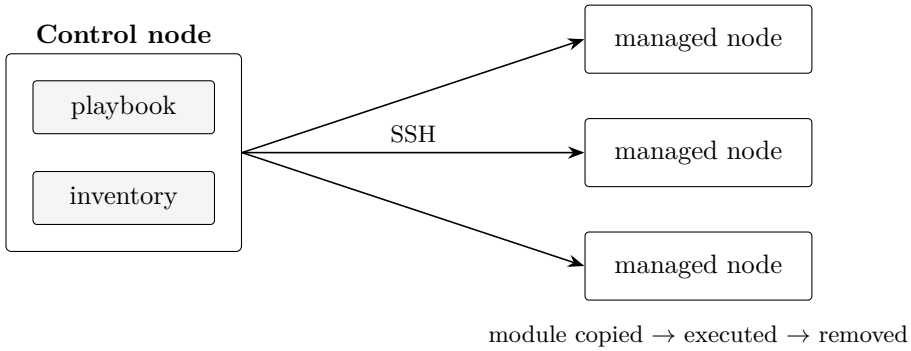


Figure 3.1: The path of one task, from the control node to the managed nodes.

Which machines a run touches comes from the inventory, a list of managed nodes produced by one or more inventory sources. Hosts are collected into groups, and variables attach either to a single host or to a group, so a play can address `webservers` without naming its members. A **static inventory** is a file in INI or YAML format, edited by hand. A **dynamic inventory** is a script or a plugin that the engine calls at the start of the run and that returns the same structure, computed from an external source such as a cloud API. The difference is that with a static file an operator maintains by hand the record of machines that some other tool created.

A **playbook** contains plays, and a **play** maps a set of managed nodes onto a list of tasks. A **task** states the action to apply to the managed host, naming a module and the arguments it takes, while a **role** packages tasks, handlers, variables, templates and files so that the same behaviour can be attached to another group of hosts [34]. Order is part of the meaning here. Tasks run from the first to the last, the hosts of a play advance through them together, and a handler executes at the end of the play only if a task that notified it reported a change [36]. A playbook therefore states a sequence of operations, even when every single task is phrased as a property the machine ought to have.

The **glossary** defines an idempotent operation as one whose result after a single execution is the same as after repeated executions with nothing in between. In Ansible that property belongs to the individual module, which reads the current state of the object it manages and acts only on the difference, reporting `ok` when it changed nothing and `changed` when it did; the engine neither computes the difference nor verifies the claim. A module that runs an arbitrary command cannot read the state of whatever the command touches, so it executes on every pass unless the task guards it with the `creates` or `removes` arguments, which skip it according to the existence of a given file. Check mode, requested with `--check`, has the same shape: modules that support it report the changes they would have made, and modules that do not report nothing and do nothing.

What the engine does not keep is a record of what it created. Nothing survives

between two runs: the description is the playbook, the observation is whatever the modules report while they execute, and there is no third term against which the two could be compared. Deleting a task from a playbook does not undo its actions, because no association exists between the task and the object it produced, so a package installed by a task that is later removed stays installed. This bounds what the tool can be asked. It reports what changed during a run, and it cannot report how the machines drifted from their description since the previous one, nor which of the objects on them are its own.

The objects the modules act on are packages, services, files, users and the contents of configuration files, on machines that already exist and keep their identity from one run to the next. Ansible is the layer underneath any cloud API, and describes the inside of a machine, not the set of machines that should exist.

3.2 Terraform

Terraform is a provisioning tool developed by HashiCorp. It creates, changes and destroys the objects a remote system exposes through its API, machines, networks, disks and accounts among them, working from a description held in files instead of from commands issued one at a time. It is written in Go, and its first tagged release dates from July 2014.

The description is declarative: the operator writes the resources that must exist, and the tool computes the operations that produce them together with the order in which those operations have to run. It is written in the HashiCorp Configuration Language (HCL), whose blocks name a resource type, a local identifier and the arguments the resource takes. Among the 44 companies interviewed in the industrial study cited in Section 2.3, Terraform is the provisioning tool named most often [5]. The configurations written for this work require a minimum version of 1.5.0.

The licensing terms changed while the tool was already in wide use, and the boundary falls inside the range of versions that constraint accepts. On 10 August 2023, HashiCorp moved its products from the Mozilla Public License 2.0 to the Business Source License 1.1, leaving the versions already published under the earlier terms. The releases show where the line sits: the licence shipped with Terraform 1.5.7, the last of the 1.5 series, is MPL 2.0, while the one shipped with 1.6.0 is BUSL 1.1, naming HashiCorp as licensor and MPL 2.0 as the licence the code reverts to four years after each release. A fork was created in response to the change:OpenTofu. It is hosted by the Linux Foundation and continues the MPL line and reads state files written by Terraform up to 1.5.x.

3.2.1 The declarative model and the resource graph

A `resource` block names a type, which comes from a provider, and a local name. The pair forms the address by which the resource is referred to everywhere else. Arguments go in, attributes come out, and writing the attribute of one resource inside

the argument of another records a dependency between the two. Terraform reads those references and builds a directed graph of the configuration, then walks it [37], so what the operator fixes is which objects have to exist and not the sequence in which to create them. Resources with no path between them are processed concurrently. A dependency that does not pass through a reference, because it is imposed by the remote system rather than by data, is declared with `depends_on`.

Two other block types complete the picture. A `data` block reads an object that exists in the remote system without claiming it, and the configuration can use its attributes and will never modify or delete it. A `variable` block declares an input and an `output` block a value the configuration exposes, and together they form the interface through which one piece of configuration is called by another.

3.2.2 Providers

Terraform core doesn't have any knowledge of any API. Every resource type is contributed by a provider, a plugin distributed and versioned on its own, which the CLI launches as a separate process and which translates the creation, reading, update and deletion of a resource into calls to a remote API. A configuration declares the providers it needs in a `required_providers` block, giving the source address in a registry and a version constraint. `terraform init` downloads the matching binaries and records their checksums in a lock file, so that the same configuration resolves to the same plugin on another machine.

The provider decides what can be written at all. A primitive that the remote API exposes but the provider does not model has no resource type, therefore no address, and cannot be declared. This work uses the OpenNebula provider version 1.5.0, published in the public Terraform registry and pinned to that exact release rather than to a range [38]. Chapter 6 reports where its coverage of the OpenNebula API stops.

3.2.3 State

The state file is the record of the resources under management. For each address in the configuration it stores the identifier of the corresponding object in the remote system, together with the attribute values read the last time the two were compared. It is the only means by which Terraform tracks what it manages, and it holds those values in clear, including the ones that are secret.

The tool exists to answer to two questions. Without the state record neither of the two questions would be decidable. The tool could not distinguish a resource it created from an identical one created by somebody else, since the only thing that differentiates them is the recorded association between an address and an identifier. And deleting a block from the configuration would carry no meaning, because the object that block used to describe would no longer be named anywhere. Before comparing, Terraform refreshes the state, reading back from the remote API the objects it lists and updating their recorded attributes.

The state is also the term that can be wrong while the configuration is right. In the analysis of 5,110 state reconciliation defects cited in Section 2.3, the largest category concerns the inventory itself [8]. Correct code applied against a state that misdescribes reality acts on the wrong object, or fails to act at all.

Where the file lives is a backend setting, either a local file or a remote store that supports locking so that two operators cannot apply at the same time. A state is also the extent within which a mistake propagates, since every resource it lists is a candidate for the operations of a single run.

3.2.4 The plan and apply cycle

The ordinary use of Terraform is covered by four commands:

- `init` prepares a working directory, downloading the providers the configuration requires and configuring the backend;
- `validate` checks that the configuration is syntactically well formed and internally consistent, without contacting any remote API;
- `plan` refreshes the state, compares it with the configuration, and produces the list of operations that would close the difference, each of them one of four kinds: create, update in place, destroy and recreate, or destroy;
- `apply` executes the list of operations provided by `plan`, and writes the result back into the state.

`destroy` follows the same mechanism of apply, but with the empty set as its target. Figure 3.2 shows the path a change travels, from the two terms the plan compares to the infrastructure the apply reaches through the provider.

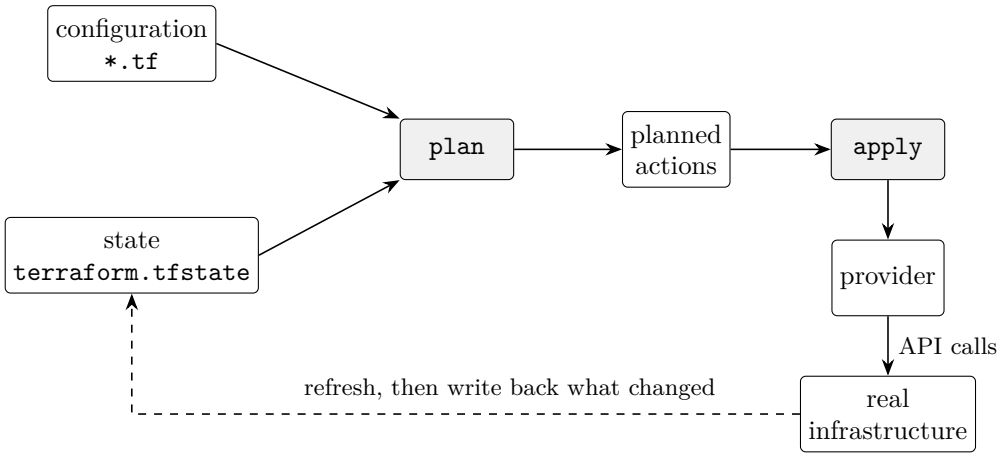


Figure 3.2: The three terms Terraform works with, and the two commands that relate them.

A plan is an artifact and can be saved to a file, and `terraform show -json` turns that file into a machine-readable document that describes the planned changes together with the configuration and the state they were computed from. This is

the property the rest of this thesis depends on: between the moment a change is fully determined and the moment it reaches the cluster, there is an object another program can read and judge. This property also carries the implicit condition that the plan is valid with respect to the state and the remote system as they stood when it was computed, so a saved plan applied later can be refused because some of the properties underneath it have changed.

3.2.5 Modules and state roots

A module is a directory of configuration files. It takes input variables, returns outputs, and can be called by another configuration as many times as needed. The directory in which the command is actually run is the root module. A called module has no state of its own, and the resources it declares live in the state of the caller, under an address prefixed by the call, which is what makes the whole tree of calls a single unit of planning. The `count` and `for_each` arguments instantiate a resource or a module once for every element of a collection, so that the number of tenants, machines or networks becomes data instead of repeated blocks.

The counterpart of that single unit of planning is that everything reachable from one root is planned, applied and destroyed together. Dividing a system into several roots trades the convenience for separation. Each root keeps its own state, and a mistake inside one cannot reach what the others hold. The price is that values no longer travel by reference, so a root that needs an identifier produced by another has to read it explicitly, through the `terraform_remote_state` data source or through variables passed in from outside. Chapter 4 describes how the framework built here is divided.

3.2.6 Configuration management and provisioning

Ansible and Terraform fall under the definition of IaC, but they answer different questions. Ansible takes a machine that already exists and brings its contents to a described state. Terraform decides which objects should exist at all and asks an API to make it so. The first assumes its target, and the second is the one that produces it.

The difference that generates all the others is the record. Terraform keeps an inventory of the objects it created, so it can compute a difference between a description and a reality and act on that difference, and the removal of a declaration becomes an operation with a meaning: destroy what the declaration used to name. Ansible keeps nothing between two runs, so the same removal means only that a task no longer runs. Convergence in Ansible rests on each module inspecting its own object at the moment it executes, which is a local property established one task at a time. Convergence in Terraform is computed globally before anything is touched.

There are two failure modes that follow from this, and neither tool covers the other's:

- Terraform can act correctly on the wrong object when its state has drifted from the remote system.
- Ansible can do different work on the second run than on the first when the modules a playbook uses are not idempotent, and nothing in the engine detects it.

One risk lives in a file while the other lives in the modules, and the boundary between the two is the API itself. Terraform creates a virtual machine and stops at its edge. The operating system inside it, the packages on it and the services it runs are on the far side of an interface Terraform does not cross, and that is where configuration management works. Table 3.1 collects the axes along which the two differ. They are used together rather than one instead of the other.

Table 3.1: Ansible and Terraform along the axes that separate configuration management from provisioning.

	Ansible	**Terraform**
Unit of work	Task applied to a managed node	Resource in a remote API
Record between runs	None	State file mapping each address to an object
Execution model	Ordered sequence of tasks	Graph derived from references between resources
Convergence	Per module, while the task runs	Computed globally before anything is applied
Removing a declaration	No effect	Destroys the object it named
Typical object	Package, service, file, user	Machine, network, disk, quota, group

3.3 OpenNebula

OpenNebula is an open source platform for building and operating private clouds. It presents the hardware of a datacenter as pools of hosts, datastores and networks, and hands parts of those pools to users in the form of virtual machines, virtual networks and disk images. It is developed by OpenNebula Systems together with a community of contributors, and distributed under the Apache License 2.0 [39]. The installation described in this thesis is the Community Edition, version 7.0.1.

Every object the platform manages is created and modified through one interface, and that interface is where the two previous sections stopped. Terraform reaches it from outside and asks for objects to exist. Ansible works inside the machines those objects become, and never addresses the platform at all.

3.3.1 Front-end, nodes and drivers

An installation has one front-end and a set of nodes. The front-end runs `oned`, the daemon that holds the record of every object under management, hosts, virtual networks, datastores, users, groups and virtual machines among them, and that exposes the XML-RPC API through which all of them are created and changed [40].

The nodes run the hypervisor and the virtual machines themselves, and carry a package chosen according to the hypervisor they run, `opennebula-node-kvm` or `opennebula-node-lxc` on Debian and Ubuntu.

The front-end drives the nodes over SSH as the `oneadmin` user. Migrations and image transfers travel on that connection, so a node needs only an account the front-end can reach, not a service that listens for it. Since version 5.12 an authentication agent on the front-end answers the authentication requests, which keeps the private key of `oneadmin` off the nodes entirely. The transport is the one Section 3.1 described, used here to reach the hypervisor instead of the guest.

Drivers are where the platform is extended, and they take the shape Section 3.2.2 described for a Terraform provider. Each one is declared in `oned.conf` as a block naming an executable and its arguments, and `oned` runs it as a separate process. `VM_MAD` drives a hypervisor, `IM_MAD` collects the monitoring data of a host, `TM_MAD` moves disks between datastores, and `AUTH_MAD` answers the question of who may do what. KVM and LXC are two `VM_MAD` entries over the same core, and a host record binds a machine to the drivers it is expected to run, so what is registered is not the machine alone.

Placement is decided by a daemon of its own, installed with the same package as `oned` and able to run on another machine. The scheduler looks for virtual machines in the pending state, discards the hosts that cannot take each one, ranks the remainder and issues the deployment. What narrows the first step is `SCHED_REQUIREMENTS`, a boolean expression over host attributes carried by a virtual machine or by its template. A machine whose expression no host satisfies stays pending rather than being deployed on a host that does not match it, and since an administrator can write attributes into a host record instead of only reading the ones the monitoring driver discovers, placement can be constrained on properties the platform itself knows nothing about.

Three kinds of client can sit on the API:

- the `one*` command line tools;
- the Sunstone web interface;
- any program that speaks the protocol, such as the OpenNebula provider for Terraform.

Wrappers are published for Ruby, Java, Go and Python. None of them enforces anything on its own, because the permission checks run inside `oned`, and the same account therefore receives the same answer whichever client asks. Figure 3.3 places the three of them, and the two tools of the previous sections, with respect to that boundary.

3.3.2 The resource model

A host is a machine registered with the drivers it runs. Its record carries the capacity the scheduler allocates from and whatever attributes an administrator adds to it.

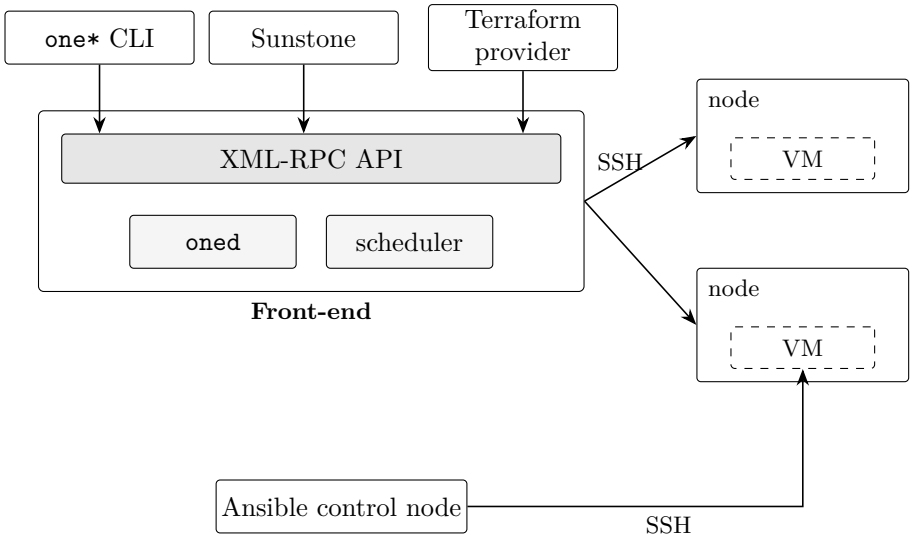


Figure 3.3: Where each tool meets the platform.

A cluster groups hosts, datastores and virtual networks that belong together, and it is the unit by which a whole set of infrastructure is granted or withheld in one operation instead of one object at a time.

Storage is divided into three kinds of datastore:

- the image datastore, which is the repository of disk images;
- the system datastore, which holds the disks of running virtual machines;
- the files datastore, which keeps contextualisation files and kernels.

How a disk travels between them is decided by the transfer driver named in `TM_MAD`, so the same image can be copied over SSH on one installation and referenced in place on another, with no change to the objects declared above it.

A template is a stored description of a virtual machine, and instantiating it produces a machine with an identifier of its own. Attributes can be added at instantiation, so one template serves several sizes. The `CONTEXT` section of a template is how values reach the inside of the guest. The platform writes them into a small ISO image, attaches it to the machine as a disk, and a package installed in the guest reads it at boot and applies what it finds. Network configuration and the authorised public key arrive that way, through `NETWORK` and `SSH_PUBLIC_KEY`, which is the seam through which Ansible later reaches a machine the platform created.

A virtual network names a driver in `VN_MAD` and one or more address ranges. The driver decides how separation is realised on the wire, with entries for a plain bridge, for 802.1Q tagging, for VXLAN and for Open vSwitch, while the address ranges decide which addresses exist to be handed out at all. A network interface attached to the network takes a lease from one of the ranges, and that lease is what contextualisation writes into the guest.

A virtual router is an object with interfaces on more than one virtual network. It is instantiated from a template like any other machine, and what runs is a small

appliance that forwards between the networks its interfaces sit on. It is how two networks the platform otherwise keeps apart are made to exchange traffic without giving a guest an interface on both.

3.3.3 Users, groups and the primitives of confinement

Every user belongs to a primary group and may belong to others, and every object records an owner and a group. Two mechanisms then decide what a user can do, and they answer different questions:

- Permissions are attached to the object and say who may act on it.
- Access control rules are attached to no object and say which objects of a class a subject may act on at all.

Both have to be right, and one of them being right does not compensate for the other being wrong.

The permission field of an object is nine bits, three rights for each of three subjects. The rights are USE, MANAGE and ADMIN, worth 4, 2 and 1, and the subjects are the owner, the group and everyone else, so the field is written in octal exactly as a Unix mode is [41]. For example, a virtual network with permissions `640` can be used and managed by its owner, used by the members of its group, and neither seen nor used by anyone outside it. USE covers reading an object and consuming it, MANAGE covers changing it, and ADMIN covers the operations reserved to administrators.

An access control rule has four parts: the subject it applies to, the set of objects it covers, the rights it grants and the zone it holds in.

A subject is a user, a group or everyone. An object set is named by identifier, by owning group, by cluster, or as a whole class. While the permission field answers a question about one object that exists, a rule answers it about a set described in advance, and therefore also about the objects of that set which have not been created yet.

A VDC binds a set of groups to a set of resource providers, a provider being a host, a datastore, a virtual network or an entire cluster, taken within a zone. Adding a provider to a VDC writes the access control rules that grant its groups the corresponding right, with the rights themselves configured in `oned.conf`, where the defaults give MANAGE over a host and USE over a network or a datastore. Removing the provider withdraws those rules. What a VDC adds is a name for a set of grants and one place to change it, which is what makes the set issuable and revocable as a unit.

Quotas limit consumption but not access, and they attach to a user or to a group. There are four families:

- Compute quotas, over processors, memory, number of machines and system disk size.

- Datastore quotas, over stored size and number of images.
- Network quotas, over the addresses leased from a given network.
- Image quotas, over how many machines may run from a given image at once.

Every limit is paired with a counter that moves as objects are created and destroyed, and two values are special: `-1` for the default quota and `-2` for no limit. Changing the quotas of a group requires ADMIN over it, which by default only `oneadmin` holds, so a group cannot raise its own ceiling.

3.4 Natural Language Interface Stack

The three tools described until here carry out what is written in files or commands. A language model, instead, produces a likely continuation of its input, and nothing inside it separates an answer that is correct from one that merely reads well. There are some components around it that can be used to narrow what it is allowed to produce and to turn that product into something another program can check.

Three of those parts are described below:

- A server that runs the model.
- A framework that presents operations to it and collects what it emits.
- A schema layer that decides whether what it emitted is well formed.

A backend and an interface carry the conversation between an operator and those three parts. In chapter 6 it is described how they are wired together for this work.

3.4.1 Serving a model locally

A language model takes a sequence of text and produces a continuation of it, one token at a time. Running one means holding several billion parameters in memory and performing an arithmetic pass over them for every token produced, so the two factors that decide whether a given model is usable are the memory its parameters occupy and the speed at which that pass runs on the hardware at hand.

Ollama serves a model taking care of both factors. It is written mostly in Go, with the inference kernels in C and C++, and distributed under the MIT licence [42]. It downloads a model named by a tag, keeps it loaded between requests, and exposes an HTTP API on port 11434 through which a client sends a conversation and receives the reply. Weights are stored in the GGUF file format, which holds them together with the metadata a runtime needs in order to load them.

Quantisation is what makes a model of this size fit in consumer electronic devices. Rather than holding every weight as a 16-bit float, a quantised model stores it in fewer bits, four of them in the `q4_K_M` scheme that is among the levels `ollama create` can produce. A quantised model runs faster and consumes less memory, but with reduced accuracy [43]. This trade-off is what allows a model with billions of parameters to be

served from the memory of the same front-end that runs `oned` with relatively little impact on its performance.

A small language model is one that "can fit onto a common consumer electronic device and perform inference with latency sufficiently low to be practical when serving the agentic requests of one user" [44]. The same authors add that in 2025 they would place most models below ten billion parameters in that class. What the definition makes small is the pairing of a model with a deployment rather than the model alone. The interface serves one operator at a time, on a machine that exists for other reasons, and a model is small when it runs there.

Running the model on that machine is a requirement. Requirement 4 of Section 2.2 rules out a service outside the premises, and a managed model API would carry a description of the infrastructure off site with every request. However, there are some consequences when serving locally. Generation is sampled, temperature controls how far the sampling strays from the most likely token, and setting it to zero is not on its own enough to make two runs of Ollama identical, since the sampler also takes a seed. Both are parameters of the request, so the same loaded model can be driven deterministically or not without being reloaded.

3.4.2 Tool calling

A model that replies in prose is not suitable to use directly in front of infrastructure. The reply has to be parsed and a parser for free text is a source of errors. To avoid these errors, tool calling is generally used to make the model produce a standardized output.

Alongside the conversation, the request carries a list of the tools the caller is willing to run. Each entry holds a name, a description in natural language, and a JSON Schema of the arguments the tool takes, giving their types and which of them are required. The model may then answer with a tool call in place of a message, naming one of the tools and supplying an object of arguments for it.

This takes the model to a closed output space. It does not write Terraform and cannot invent an operation that does not exist, since the only thing it can emit is the name of a tool it was given together with values for that tool's declared arguments. This is the second of the two families of Section 2.6, and it confines what can go wrong to the choice of tool and the value of an argument, both of which can be inspected before anything runs.

The arguments of each operation are declared as a Pydantic model, a Python class whose fields carry types and defaults, and the object the model produced is instantiated against that class. A value of the wrong type, or a required argument left out, fails there and never reaches the operation. What a schema cannot catch is an argument of the right type and the wrong value, which is the reason the policy check and the operator's reading of the plan exist at all.

LangChain is the layer between the model and the tools. It is an open source framework for building applications on language models, developed by LangChain

AI and distributed under the MIT licence [45]. Three of its parts are used here:

- a chat model class that wraps the Ollama endpoint behind the interface every other backend is given, so the backend is a configuration value;
- `bind_tools`, which attaches the tool declarations to that model and translates each Pydantic schema into the JSON Schema the request carries;
- a reply object that exposes the calls already parsed, as a name and a dictionary of arguments, so the application reads a structure instead of text.

Figure 3.4 shows the round trip and the two shapes a reply can take.

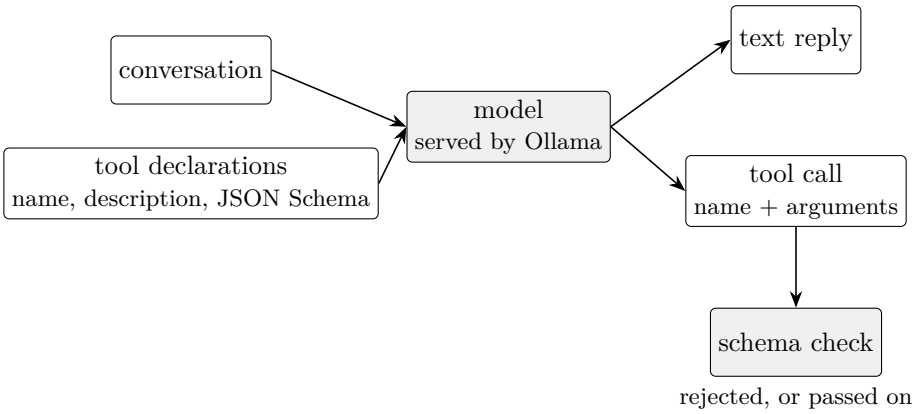


Figure 3.4: The two shapes a reply can take.

In this work, the models are used as published. Nothing is done to train or fine-tune them, so what a given model contributes is the ability it already had to follow a schema, and replacing one with another is a change of tag in the configuration file.

Chapter 4

System Architecture

4.1 Overview

The framework is the component that stands between an operator who decides that a tenant should exist and the OpenNebula front-end that makes it exist. What it puts in that space is a chain of artifacts, each of which can be read, judged or refused before the next one is produced.

The rule that holds across the whole design is that a change can reach the cluster only as a Terraform plan that a machine and a person have both accepted, and the path the automation takes contains no branch around either of them. Everything that follows in this chapter is a consequence of this rule, either a component that produces the plan, one that judges it, or one that records what was decided.

This procedures leave behind a running tenant and a set of registers compiled after its execution. The design leaves behind three artifacts:

- The **declaration** states which objects a tenant is made of, in files kept under version control. It is written once and applied as many times as needed. Removing an entry from it is the operation that decommissions what the entry named.
- The **plan** lists the operations that would close the difference between the declaration and the cluster as it currently stands. It is computed before every change, and every check applies to it.
- The **decision record** says who accepted a given plan and on what grounds. It collects the report of the policy evaluation, the operation and arguments the operator approved, and, when a tenant is decommissioned, an audit bundle exported before the first destroy.

The registers used by an operator to compile by hand are a serialisation of the first and the third.

The boundary of the system is drawn at the front-end API on one side and at the guest operating system on the other. Inside it are the declaration of a tenant, the identities, networks, storage and machines that realise it, the evaluation of the

policy set, the approval step, the export of audit material, and the natural language interface through which all of the above can be driven. Four things stay outside:

- bringing a bare machine to the state in which it runs OpenNebula, which is the deployment of the platform itself;
- the configuration of the operating system inside a guest once the guest has booted;
- the backup of a tenant's machines and the custody of its encryption keys, which remain the manual procedures for the scope of this thesis;
- whatever the customer does with the machines it was given.

Figure 4.1 places those four outside a box that contains everything else.

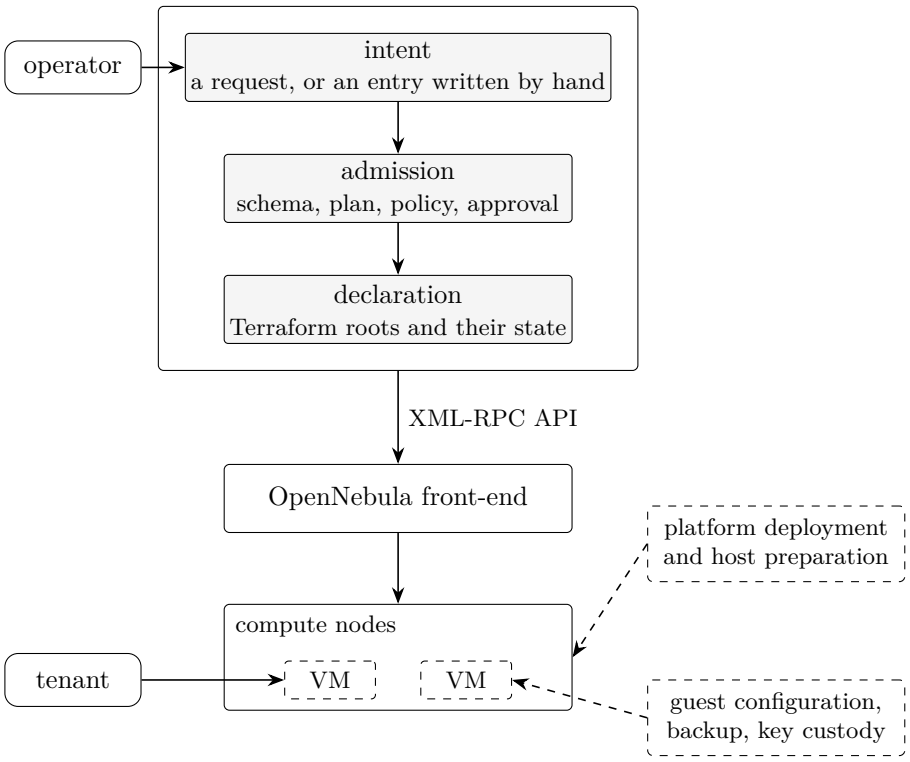


Figure 4.1: The boundary of the framework, and what is left outside it.

Only operators drive the framework. A tenant holds an account on the platform, consumes the resources allocated to it, and never sees the code that created them. Enforcement stays with the platform, which applies its own permission and access control checks to every call it receives. An operator holding administrative credentials reaches the same API through the command line tools or the web interface, so the checks described here sit on the path the automation takes rather than in front of the API.

4.2 Architectural Layers

The system decomposes into two main directions. Along the first, a change descends from a sentence typed by an operator to a machine running on a host, and every step down accepts a narrower vocabulary than the one above it. Along the second, the description of the infrastructure is cut into four independent units, each holding its own record of what it manages.

4.2.1 The vertical stack

When a request is typed in natural language, there are five layers that separate it from its effect. Each one is defined by what can be expressed in it and by what it hands to the layer below:

1. The **intent** layer takes a request in natural language, or an entry an operator edits by hand in a database. It produces the name of one of six operations that can be performed, together with a set of arguments.

2. The **admission** layer decides whether the operation chosen may run. It instantiates the arguments against the schema of the operation, computes the plan in a dry run, evaluates the policy set over a document built from the databases, and asks the operator to approve. It produces a decision on one specific plan.

3. The **declaration** layer holds the configuration and the state, that is, which objects should exist and which of the existing ones are under management. It produces calls to the front-end API.

4. The **platform** layer is `oned`. It resolves each call against the permissions of the objects it names and the access control rules of the account that issued it, and produces the objects themselves.

5. The **substrate** layer is the hosts, the hypervisors, the bridges and the datastore backends on which those objects become real. The framework does not reach it. The front-end drives it over SSH, and the machines are brought to the state in which they can be driven by the deployment of the platform, which is a separate procedure.

The vocabulary shrinks at every step. An operator may type any sentence, the intent layer turns it into just one operation name and a set of arguments, the declaration layer knows only the resource types the OpenNebula provider models, and `oned` accepts only the objects its own API defines. That reduction is what makes a natural language interface tolerable in front of infrastructure. The model never writes a configuration from scratch, so the surface on which it can be wrong is just a choice among a small number of operations, and the values of a handful of arguments; both are visible in the plan before anything is applied.

Each check sits where the corresponding error becomes detectable:

- An argument of the wrong type, or a required one left out, fails against the schema of the operation and never produces a plan.
- An argument that is well formed and forbidden fails against the policy set, which covers a processor count above the tenant's quota, a host outside the tenant's allowlist, and a base image outside the approved catalogue.
- A call whose account does not carry the right fails inside `oned`, whichever client issued it.

However, none of the three catches an argument that is well typed, permitted by the policy set, and different from what the operator meant. This semantic error is what the approval step answers for, and it is the last thing that happens before an apply. Figure 4.2 shows the five layers with what each one accepts and what it passes down.

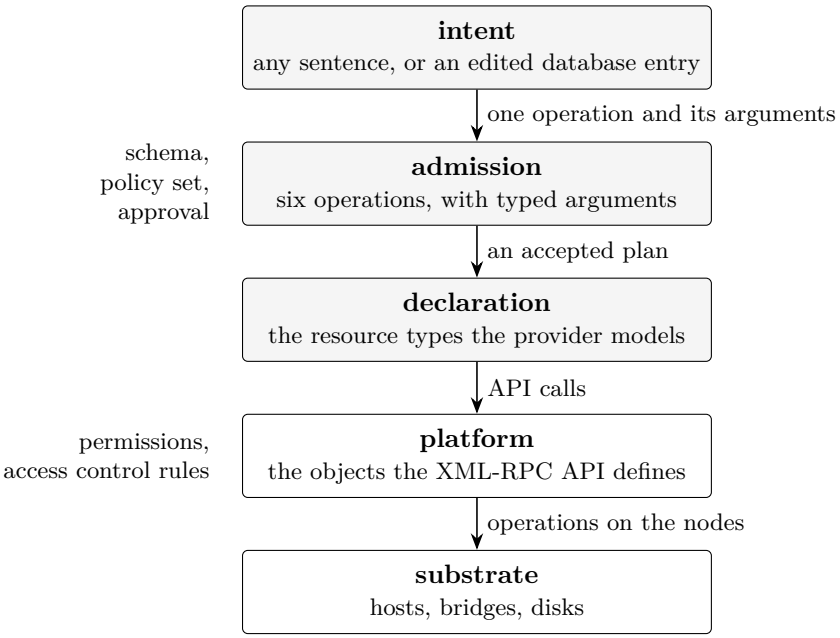


Figure 4.2: The five layers, what each accepts, and what it hands down. The labels on the left name the checks a layer runs on what it receives.

4.2.2 Partitioning the declaration

Everything reachable from a Terraform root is planned, applied and destroyed as one unit, so the size of a root is the size of the smallest change that can be made in it. A single root for the whole installation would make the addition of one customer a plan over the management network, the shared images and every other customer already provisioned. We cut the declaration into four roots, each with its own state:

- `01_platform` holds what every tenant shares. It creates the management network and its address range, the image and the instance template of the virtual router, the shared base images and templates, and the template that carries the real TEE and GPU configuration. It also imports the default VDC

that OpenNebula ships with, in order to strip the grant that would otherwise give every group access to every resource.

- `02_tenant_management` holds the tenants. It instantiates one composition module for every key of the file `tenants.json`, and each instance creates all the resources needed by a tenant: the group, the users and their quotas, the VXLAN network and its security group, the virtual router that attaches that network to the management one, the image datastore, the clone of the base image into it, and the VDC that binds the group to exactly those resources.

- `03_workloads` holds the machines that run inside a tenant. It reads the identifiers of the group and the network from the state of the layer above and takes only the sizing from `workloads.json`.

- `vm_instances` holds single machines deployed onto infrastructure that already exists, one entry of `vms.json` for each. The virtual machines can be tied to a tenant or just use the shared infrastructure.

There are three criteria that decide where the lines fall. The first is the rate of change, since the platform is applied once and a tenant is applied whenever a customer is added or removed. The second is how far a mistake can travel, since a root is also the extent within which a wrong plan can act. The third is who drives the change, since `02_tenant_management` is the root the natural language interface writes to, while the platform root is applied by an operator working directly on the files, at the beginning of the installation process.

The separation runs between the platform, the tenants and their workloads. It does not run between one tenant and another. An alternative option would have been to use one root per tenant, but we discarded it for many reasons. Each root would need its own initialisation and its own state file, growing with the number of customers. Every one of them would have to read the platform identifiers through a remote state data source, turning a single dependency into as many as there are tenants. Finally, the tenant database would no longer be one file iterated by a single configuration, which is precisely the shape the interface of the intent layer edits.

With this architecture, values travel between roots explicitly. A root that needs an identifier produced by another root reads it through a `terraform_remote_state` data source pointed at the state file of the root above, so each dependency is written in the configuration rather than left implicit in a reference. Those data sources also fix the order in which the roots are applied, from the platform to the tenants and then to their workloads, and the scripts and tools that carry out an operation apply the layers in that order on the operator's behalf.

Agreement between the databases is established while the plan is computed. `03_workloads` declares preconditions that stop the run and name the offending entry when `workloads.json` refers to a tenant that the state of `02_tenant_management` does not hold, so a mismatch surfaces before any call reaches the front-end. The check runs in one direction by design. A tenant present in layer 02 with no entry in

layer 03 is accepted, which is what lets offboarding remove the workload entry first and destroy the machines alone, leaving the identity and the network to follow in a second apply.

`vm_instances` sits outside the chain, as it doesn't read any remote state and it resolves the network, the datastore and the group it needs from its own variables, so it can be applied at any time and with no tenant declared at all. The same root can serve a machine on shared infrastructure or an additional machine inside an existing tenant.

The three databases are the seam between the intent layer and the declaration layer. `tenants.json`, `workloads.json` and `vms.json` are the only inputs the configurations read, and the configurations themselves are fixed. This is what keeps the intent layer from having to produce HCL. Figure 4.3 shows the four roots, the database each one reads, and the state each one reads from the others.

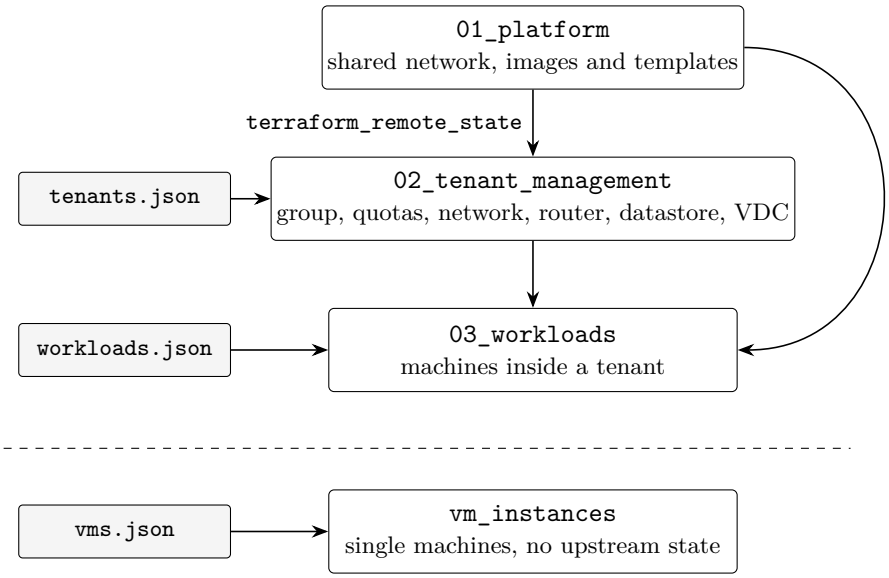


Figure 4.3: The four state roots, their inputs, and the dependencies among them. The dashed line separates the ordered chain from the standalone root.

4.3 Components and Responsibilities

In this work, responsibility is divided so that the component which decides what to do never reaches the platform, and the one which reaches the platform decides nothing. The interface collects a request and returns a plan, the model picks one operation and fills its arguments, the operation edits the databases and drives Terraform, the judges answer a question about the state the operation proposes, and know nothing of how it was arrived at. Each component hands its result to the next one, and none of them reaches past it.

4.3.1 The interface

There are two entry points that lead to the same operation. The first is a chat interface, where an operator writes a request and reads what comes back. The second is a terminal, where an operation is invoked as a script with its arguments on the command line. The second one is the path an operator takes when a request is easier to type as arguments than as a sentence, and the only one available when the model is not running.

The interface owns the pairing between a plan and the answer given to it. It holds one plan awaiting a decision at a time, presents it, and carries an approval or a refusal back to the operation that produced it. It does not compute, evaluate or issue any call to the front-end, so a defect in it can lose a decision or show the wrong text, and cannot create a resource.

A plan may be shown in a more compact form, like the one for the offboarding. The operator reads the totals and the name of every resource that would be destroyed, rather than the whole Terraform output. What the compaction may not do is drop a resource from that list, because the list is what the operator is judging.

4.3.2 The agent

The agent is a language model with the six operations bound to it as callable schemas, and a prompt stating the rules it must follow when it fills them. It returns the name of one operation and a set of argument values. It never writes HCL, never names an OpenNebula object, doesn't run any code.

The argument values are instantiated against the schema of the chosen operation before any other step. The schema states the type of each argument and the consistency rules between them, so a required value left out, a string where a number belongs, and a quota smaller than the workload it is meant to contain all fail at this point. The operator reads the reason in the same conversation, without a plan being produced and no file being touched.

The model runs on the same installation as the rest of the framework, served locally, so a request naming a customer stays inside the perimeter. One operation, `get_state`, is unlike the other five. It reads the databases and the state files and returns what is deployed, so it produces no plan and asks for no approval, and its answer is streamed straight back to the operator.

4.3.3 The operations

An operation is the unit in which the rest of the system is addressed. In this architecture, there are six of them, and each one exists in two forms that run the same steps in the same order: a script an operator invokes from a terminal and a wrapper the interface calls with the arguments the model produced.

Every operation computes what the change would do and returns it, leaving nothing behind that survives the call. Then, it carries the change out and takes the operator's approval as an argument, refusing to run when it is absent. Splitting the

two is what makes the approval a step rather than an option, since there is no third call that applies without one.

There are three things that are known to the operation: which databases the change touches, in which order the roots are applied, and which gates it must cross.

Most of the values a tenant needs in order to exist are derived. An onboarding request usually names the customer and, at most, the size of what it should run. The operation reads the tenant database and takes the first VXLAN identifier, subnet, gateway and router management address that are not already in use. A value the request names explicitly is refused when another tenant holds it, and a key that is already present is refused. Table 4.1 lists the six operations. All of them except `get_state` pass the schema, the policy set and the approval, so the fourth column names only what a change crosses in addition to those.

Table 4.1: The six operations, with the databases each one edits and the roots it applies.

Operation	Databases	Roots	Additional gates	Destroys
`get_state`	none	none	none	no
`onboard_tenant`	tenants, workloads	02, then 03	attestation, for a TEE workload	no
`offboard_tenant`	tenants, workloads	03, then 02	audit export, and the tenant key typed by the operator	yes
`scale_tenant`	workloads	03	attestation for a TEE workload, and the tenant key typed on a reduction	on a reduction
`update_quota`	tenants	02	none	no
`deploy_vm`	vms	`vm_instances`	none	on a resize

4.3.4 The judges and the record

The policy engine receives one document assembled from the three databases and from three files holding the values the rules compare against, which are the hosts, the base images, and the bounds a machine has to fit in. The rules are grouped in four families:

- **Quotas**: the workload declared for a tenant must fit the quota declared for the same tenant, on processors, on memory and on the number of machines.
- **Residency**: every tenant names at least one host it may run on, and a workload asking for a TEE names a host that offers one.
- **Image provenance**: a machine boots only from an image in the approved list.
- **Standalone machines**: an entry of the third database stays within the declared bounds on processors, memory, disk and count, and attaches only to a network on the allowlist.

The document holds all three databases whatever the change was, and every rule runs over it, so a violation left in one database blocks a change asked for in another. The layer that asked is recorded and takes no part in the decision.

A database that does not exist enters as an empty object, which is a database with nothing to check. A policy data file that does not exist enters as a null, and the rules test for it and deny. Losing the file that lists the hosts a tenant may use blocks every apply, instead of admitting every placement.

The verdict is a list of messages, and each message names the entry and the rule that refused it. A record of the evaluation is written whatever the outcome, so a run that passed leaves the same kind of evidence as one that was blocked. The rules carry their own unit tests, and the continuous integration pipeline runs those tests and evaluates the whole policy set over the databases at every change to a policy or to a database, which puts a violation introduced by an edit in front of a reviewer before any operator asks for the change.

A second judge, the attestation, sits in front of the workload layer and answers a different question. Where the policy set reads what was asked for, attestation reads the state of the host that would run it. When a workload declares a TEE, the apply is blocked unless that host produces a quote that verifies. In Chapter 5, we will discuss what that quote contains and what a simulated one can and cannot show.

There are three registers that are left behind, that answer different questions:

- the record of each policy evaluation, which says what was judged and against which rules;
- the audit bundle, assembled before an offboarding from the entity dumps taken from the front-end, the lines of the front-end log that name the tenant, and the slice of Terraform state that describes it;
- the history of the databases under version control, where an accepted change is a commit, next to the structured log each operation writes as it runs.

Between them they carry what was asked, what was judged, what was applied and when. After a tenant is offboarded, the bundle is the only one of the three that still holds its identifiers, so it must be assembled before offboarding.

4.4 Interaction Flows

4.4.1 Onboarding a tenant

The agent names the operation and fills its arguments, and the operation derives the values the request left unsaid. Both entries are written into the databases before anything is computed over them, so the plan and the policy evaluation read the same files and describe one proposed state. If the change does not go through for any reason, the files are restored to what they were.

A plan says which objects would be created, and a verdict says whether any rule refuses them. A denial ends the flow there, with the databases restored and no approval asked for.

An approval refers to one plan and not to the request that produced it. A new request arriving in the same session discards the plan still waiting instead of carrying its approval over, so the artifact that gets applied is the one that was read.

The applies then run in the order the operation holds, layer 02 and then layer 03, with the attestation gate in front of the second one when the workload declares a TEE. What is left at the end is two entries committed to version control, the identifiers written into two state files, and a tenant whose machines are running. Figure 4.4 follows one request through these steps.

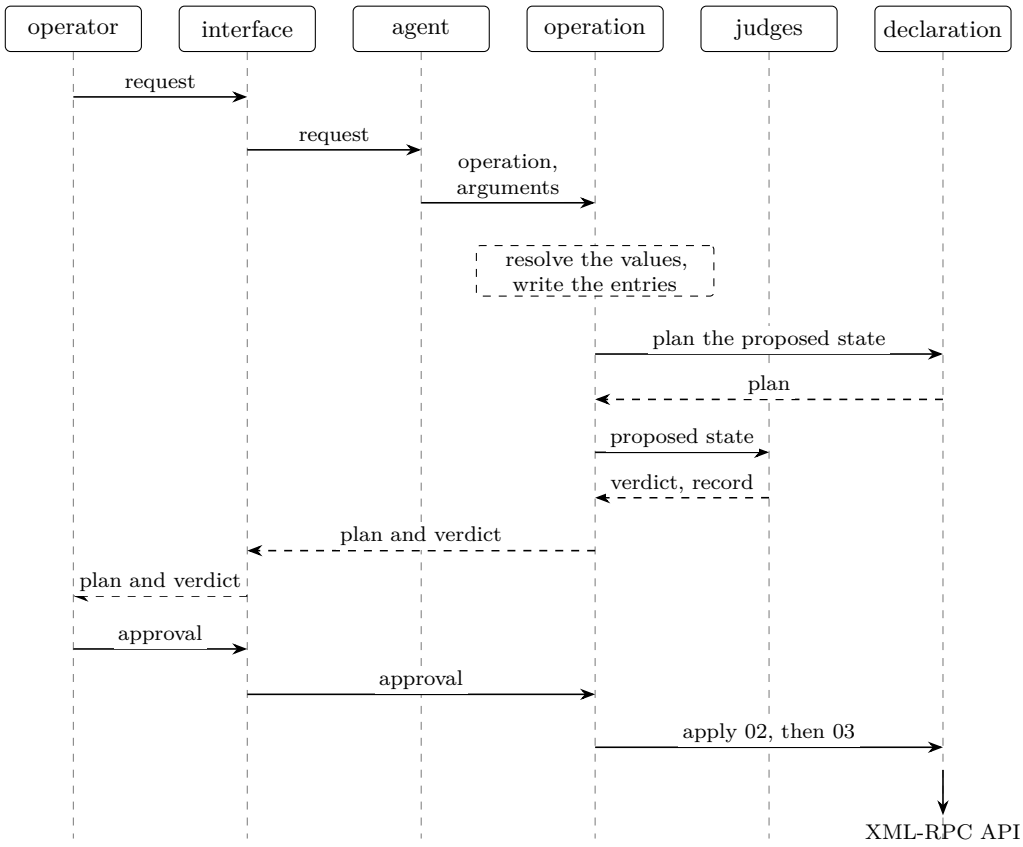


Figure 4.4: One onboarding request, from the sentence to the applies.

The same sequence runs from a terminal with no model in it. The operator invokes the script with its arguments, reads the plan Terraform prints, and confirms at the prompt. The steps between the arguments and the apply are the same ones in the same order, because they belong to the operation rather than to whatever called it.

4.4.2 Offboarding a tenant

Removal of a tenant runs in the opposite order to creation. The workload entry is removed first and layer 03 is applied, which destroys the machines. The tenant entry

is removed next and layer 02 is applied, which removes the identity, the network, the router, the datastore and the image clone. Machines hold the identifiers of the group and of the network they were created with, so taking those away first would leave objects whose owner no longer exists. The precondition of Section 4.2.2 runs in this direction for this reason, since it is what makes a tenant with no workload left legal.

The audit bundle is assembled before the first destroy. Two of its three sources stop answering as the sequence advances, because the entity dumps read objects the front-end no longer holds once layer 02 is applied, and the state slice describes resources the first apply has already removed. A bundle taken at the end would hold the tenant's own database entries and little else.

The form the approval takes is the tenant key typed at the terminal on that channel, and the script does not proceed until it matches. In the chat path the approval has already been given at the interface, and the operation answers the prompt on the operator's behalf.

4.4.3 Changes to a running tenant, and standalone machines

Resizing the workload of a tenant touches one database and applies one root. Raising the machine count adds new machines to the infrastructure, while lowering it destroys the ones that go, so the operation asks for the tenant key to be typed first, the same confirmation offboarding uses. Changing processors or memory on machines that already run replaces them rather than adjusting them in place, and the plan names each machine that would be destroyed, which is the difference the operator is being asked to accept.

A quota change touches the tenant database and applies layer 02. No machine is created or destroyed, because a quota bounds what the tenant may ask for next rather than what it already holds. A quota lowered below the workload already declared is refused by the policy set before the plan is shown, which is the same constraint the schema applies to the two arguments of an onboarding request, checked here against the databases instead of against one message.

The standalone root is reached by its own operation, reads its own database, and is judged by the family of rules written for it. It applies no other root and reads no state from one, so it runs whether or not any tenant exists. An entry naming the group and the network of an existing tenant places the machine inside that tenant, which is how a customer gets one more machine without going through the workload layer.

None of these three produces an audit bundle, since the tenant survives the change and its description can still be read from the front-end afterwards. A scale-down is the one case that destroys machines without leaving a bundle behind, and what records it is the plan the operator approved together with the copy of the previous state that the apply writes.

4.4.4 When a check refuses

There are four points in the sequence where a change can be refused:

1. The **schema** refuses before a plan exists. Nothing has been written, and the operator reads which argument was missing or inconsistent.

2. The **policy set** refuses with the list of violations, each naming the entry it read, and the record of the evaluation is written. The databases are restored and the operator is not asked to approve.

3. The **operator** refuses the plan. The databases are restored and the refusal is recorded in the session, which distinguishes a change that was rejected from one that was never asked for.

4. The **apply** fails part of the way through. Terraform's state holds what was created up to that point, and the databases keep the entry.

The first three leave the cluster untouched. The fourth is the only one that can leave it between two states, and the databases are deliberately not restored. An entry removed while the objects it named exist would put them outside every record, and the entry is also what the next plan reads in order to reconcile the difference. This is also why the order of the applies belongs to the operation. A partial onboarding leaves an identity and a network with no machines, which the next apply completes, whereas the reverse order would leave machines whose group has already been removed. Figure 4.5 places the four points on the path and names what survives each of them.

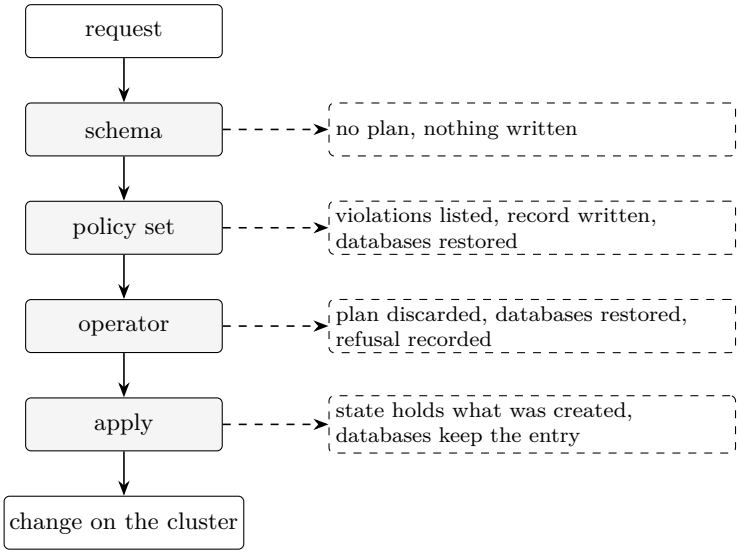


Figure 4.5: The four points at which a change can be refused, and what each refusal leaves behind.

4.5 Integration with OpenNebula

4.5.1 What the platform decides

The framework reaches the front-end as a member of `oneadmin`, the group that has every grant in OpenNebula. A narrower identity couldn't create a group, a VDC or a datastore. The platform doesn't refuses any request of the framework, and everything that limits what it may create sits above it, in the objects the declaration names and in the gates a change crosses before an apply.

Nothing in the API binds a set of calls into a single unit. One tenant is sixteen resources in the declaration before it holds a machine of its own, and each of them reaches the front-end as its own call, resolved and answered on its own. The front-end does not have knowledge of a tenant as a whole, so nothing on that side undoes a half-created one.

The host a machine runs on is chosen by the platform. What the declaration writes into a machine is a requirement expression, `SCHED_REQUIREMENTS`, which names the properties a host must have rather than the host itself. The scheduler picks among the hosts that publish those properties, and a machine whose requirement do not satisfy any host stays pending. The choice is made again every time the machine returns to the pending state.

4.5.2 What the framework declares

The declaration covers the objects a tenant is composed of. Hosts and clusters are front-end objects like those, but the declaration does not name them. They describe hardware that is already installed, and they change when the installation changes rather than when a customer arrives, which is the event the whole declaration is built around.

The declaration states that a machine exists, has a certain shape, has a certain number of processors, memory, disk, hsd a network it attaches to and a group that owns it. What moves a machine between its states stays in the platform, and that covers powering it off, suspending it, undeploying it, migrating it to another host and growing a disk in place. Those operations leave the declaration untouched.

The isolation of a tenant is declared as one VDC, which carries the lists of hosts, datastores and networks the tenant's group may reach, and from which the platform derives the access control rules that grant them. We considered declaring those rules one at a time and discarded it, because a tenant would have become a set of rules to keep aligned by hand, and removing a tenant would have meant removing every rule one by one. A VDC is created, read back and destroyed as a single object.

The default VDC is the one object the framework manages without owning. Two of its lists are declared out of scope: the groups it contains and the resources its zone grants, because the platform writes to them on its own. Every new group is added to that VDC at onboarding, and a plan that removed the entry would find it back at the next one. The grant that made such a membership worth something

was removed when the object was imported, and the platform does not restore it. Table 4.2 says who creates each object, including the three that the framework does not create.

Table 4.2: Who creates each object the system depends on.

Object	Created by
Hosts, their registration on the front-end and their labels	the deployment of the platform
The device permissions a node needs for a real enclave	an operator, by hand, once per node
Management network, shared images and templates, and the default VDC	`01_platform`
Everything a tenant is made of	`02_tenant_management`
The machines of a tenant	`03_workloads`
Machines outside any tenant	`vm_instances`
The configuration inside a guest	the contextualisation package at first boot, from values the declaration writes

4.5.3 Where the deployment stops

The deployment ends with the hosts registered on the front-end, and the declaration starts at the first call to the API. The two sides work on different units of change, since what the deployment adds or replaces is a package and a configuration file on a host, and what the declaration adds or removes is an object in the front-end's database. The deployment uses `one-deploy`, the Ansible playbook collection the OpenNebula project publishes, used with an inventory that describes our hosts.

The boundary runs between objects, since the deployment reaches the front-end as well: a host is registered through the same API, and the same playbooks can create a management network of their own. We keep the management network in the declaration because it is the network every tenant's virtual router attaches to, and a change to it has to appear in a plan an operator reads before any tenant is built on top of it.

The deployment stamps the label `TEE_TYPE` on the host, the declaration writes a requirement naming that same label, and the scheduler matches the two. Neither side reads the other. The policy set compares a workload against the list of hosts the tenant may use, which is another file in the repository, so the comparison runs between two declarations and the label the cluster publishes is never read. A workload asking for a property that was never stamped on any host passes every gate, and its machines wait in the pending state.

Part of what a tenant depends on is not an object at all. Three conditions have to hold on a node before a guest can use a real enclave:

- the platform's user on that node belongs to the group that owns the enclave device nodes;
- those devices appear in the allowlist of the hypervisor's own configuration;
- the mandatory access control profile permits access to them.

None of the three is reachable through the front-end, which holds a host as an entry in its database and never as a set of files on a machine. They were set once by hand, and they belong to the installation of a node in the same way its packages do.

The framework assumes it is the only writer of the objects it declares. Where an installation already holds one of them, that object has to be imported into the declaration or referenced by identifier and left alone.

Chapter 5

Security Model, Measures and Limitations

5.1 Tenant Model

5.1.1 What a tenant is

This thesis adopts the following definition for a tenant. A **tenant** is one customer together with the set of objects the platform holds on that customer's behalf, taken as a single unit of allocation, of authorisation and of revocation. One customer is one tenant, and the objects that belong to no tenant are listed in Section 5.1.2.

Revocation is the property that fixes the shape of the model. At the end of a contract every object of the tenant has to be found, removed and certified as deleted, which is the closing phase. We fix the kinds of object a tenant is made of, so that closing a contract is an enumeration over a known set of kinds. Each object carries either the tenant's name or its group, and the whole set can be read from the front-end.

A tenant is made of ten kinds of object:

- a **group**, which is the subject the access control rules name and the group recorded on the objects the tenant operates on;
- one or more **users**, at least one of which is a **group admin** and creates and removes the users of that group;
- a **quota** over processors, memory and number of machines, checked when a machine is allocated;
- a **VDC**, which names the hosts the group may run on, the image datastore that holds the base catalogue, and the tenant's own overlay;
- a **VXLAN overlay**, with a VNI and a private subnet of its own;
- a **security group**, carrying the rules the tenant's traffic would be filtered by;

- a **virtual router**, with one interface on that overlay and one on the management network, and it is the only object of the tenant attached to both networks;
- an **image datastore**, which may occupy 50 GB of the shared backend;
- a **base image**, cloned into that datastore, from which the tenant's machines boot;
- the **virtual machines** the tenant runs, each attached to the overlay and owned by the group.

Under the operational procedures of the lifecycle, the volumes on which a tenant's data sits are encrypted with keys of its own and those keys are held in custody. Figure 5.1 groups the ten objects divided by ownership, from the identity of the tenant to what it runs.

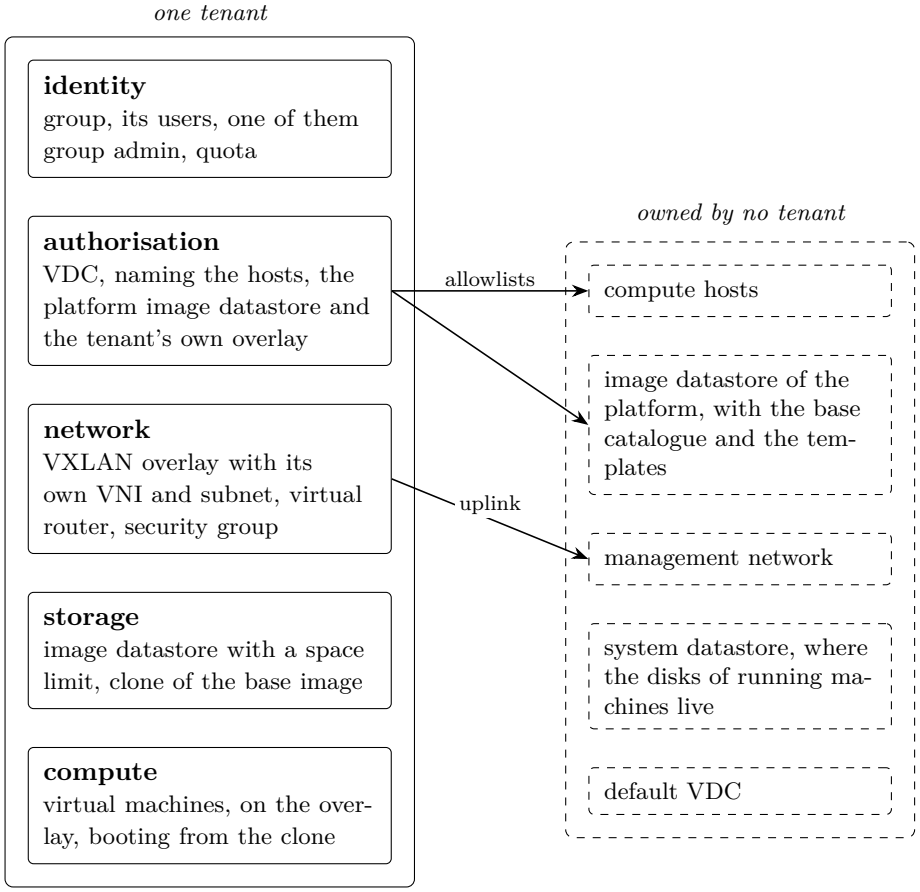


Figure 5.1: The ten kinds of object a tenant is made of, and what stays outside every tenant.

Table 5.1 maps each kind onto the OpenNebula primitive that realises it and onto the Terraform module that creates it. Every element of the model is a platform object. We discarded an intermediate representation of the tenant, computed by the framework and reconciled with the front-end, because it would add a second record to keep in step with the first, so the description of a tenant is read back with `onegroup show`, `onevdc show` and `onevnet show`. The audit bundle of Section 4.3.4 carries

those dumps for the group, the VDC, the overlay, the users and the machines, next to the entry of the tenant database and the slice of Terraform state that produced them.

Table 5.1: Each element of the model, the primitive that realises it, and the module that creates it.

Element of the model	OpenNebula primitive	Created by
Group	group	`modules/identity`
Users, group admin	user, group admin list	`modules/identity`
Quota	group quota over processors, memory and machines	`modules/identity`
VDC	VDC with one zone	`modules/tenant_onboarding`
Overlay	virtual network on the VXLAN driver, with its address ranges	`modules/networking`
Security group	security group	`modules/networking`
Virtual router	virtual router, its instance and its two interfaces	`modules/networking`
Image datastore	image datastore with a space limit	`modules/storage`
Base image	image, cloned from the base catalogue	`modules/storage`
Virtual machines	virtual machine	`modules/compute`

5.1.2 The invariants of the model

Five properties separate one tenant from another:

1. **Attribution.** The overlay, the security group, the images and the machines of a tenant carry its group. The virtual router carries the group of the platform administrator. The image datastore and the VDC carry no group at all, because neither object has a group or a permission field in OpenNebula.

2. **Disjoint overlays.** Two tenants never carry the same VNI, and their subnets do not overlap.

3. **Closed allowlist.** A tenant's VDC names hosts, datastores and networks one identifier at a time, and never the value that stands for a whole cluster. The only network it names is the tenant's own overlay, and the only datastore it names is the image datastore of the platform.

4. **One crossing to the management network.** No machine owned by a tenant's group has an interface on the management network. The one machine of a tenant that does is the instance of its virtual router, which belongs to the platform administrator.

5. **Bounded consumption.** No allocation is admitted that would take a tenant's group over the quota written for that group, and the group cannot raise that quota. A quota lowered below what the group already holds leaves the existing machines running, which is why the policy set refuses that change before the plan is shown.

There are five kinds of object that are shared and belong to no tenant:

- the compute hosts;
- the management network;
- the image datastore of the platform, which every tenant's VDC names because the base catalogue and the templates live there;
- the system datastore, where the disk of a machine is created when the machine is deployed;
- the default VDC that OpenNebula ships with.

On these five, cross-tenant access has to be refused by a mechanism.

The model does not model these three aspects:

- It does not say what happens inside a guest. The boundary is the object the front-end holds, and the operating system on the machines is administered by the customer, as it already is in the manual procedure.
- It does not let a tenant verify the state of the host its machines run on.
- It does not give a tenant a dedicated control plane. A tenant reaches the platform only through the front-end API, with an account of its own and under the rights its group carries.

5.2 Separation and Isolation Mechanisms

An invariant holds only if something refuses the call that would break it. Every refusal described here is issued by the platform while it serves a call, and none of them depends on which client sent it. The policy set judges declarations before an apply and never sees a call, so it plays no part in this section.

5.2.1 Ownership and authorisation

The two mechanisms of Section 3.3.3 (permissions and access control rules) combine as a union rather than as a conjunction, and neither can deny what the other grants. When a request arrives, the authorisation logic consults the permission field of the object named and the access control rules the installation holds, and one grant from either is enough. An object closed by its permission field is therefore not closed on its own, if the access control rules grant access.

That asymmetry was already present in the installation this work started from. When a VDC's zone names a whole cluster, the groups it holds receive MANAGE over every host of that cluster and USE over every network and datastore in it. The first version of the per-tenant VDC named the cluster that way. Each tenant overlay carried permissions `640` and, read on the object, looked closed to everyone outside its own group, while every tenant group held USE on every one of them and on the management network besides. We replaced the cluster with the three explicit lists

of identifiers, so that a tenant group now holds a rule only on the resources its own VDC names.

The VDC shipped with OpenNebula still carries the same grant. Its zone names every cluster on the front-end, and the three rules it generates give the groups it holds MANAGE over every host and USE over every network and datastore. The framework imports the object into the declaration so that it appears in a plan an operator reads, and declares an empty zone for it, but the provider does not converge on an empty value for that list, so no plan withdraws the grant. What keeps the grant away from a tenant is membership. That VDC holds one group, `users`, and the groups the framework creates for its tenants are not added to it. Membership of that one object is the part of the tenant boundary.

Four kinds of object of a tenant carry permissions `640` with the tenant's group as their group. They are the overlay, its security group, the images and the machines. The owner recorded on each of them is the account the framework applies with, not a user of the tenant, so the tenant's own users hold just what the group class carries. A user of another tenant falls in the third class and holds nothing at all, and the call it issues against the object doesn't match any rule.

The virtual router and its instance carry `640` with the group of the platform administrator: the router carries the whole of the tenant's traffic to the outside, and a tenant able to reconfigure it could give itself an interface on a network we never assigned to it. The image datastore and the VDC do not carry a group or a permission field, because version 1.5.0 of the OpenNebula provider exposes neither attribute on either resource. For the VDC the constraint agrees with the design, since a group admin editing its own allowlist would decide what its own tenant may reach.

A tenant's VDC names the hosts it may run on, the image datastore that holds the base catalogue, and its own overlay, one identifier at a time. The hosts are reachable in no other way, since a host does not carry a group of the tenant. The tenant's own image datastore appears does not apper in any VDC, so no group holds a rule on it and only the account the framework applies with writes into it. Two conditions have to hold together for a cross-tenant call to be refused: the object denies everyone outside its group, and no rule anywhere grants what the object denied. The first comes from the permission field, the second from invariant 3 and from the tenant's group belonging to no VDC other than its own.

5.2.2 Network separation

Each tenant's overlay is a VXLAN segment with a VNI of its own. Encapsulation and decapsulation run on the compute host, on the bridge the network driver creates there. A frame leaving a machine of one tenant is encapsulated with that tenant's identifier and is delivered only where that identifier is bound, so it does not reach the bridge of any other tenant. A guest chooses what it puts in a frame, but the host stamps the identifier on it.

Distinct identifiers separate the overlays on their own, and the subnets are disjoint for a second reason. Every tenant's traffic to the outside crosses onto one shared management network, where two tenants numbered from the same subnet would present the same addresses and a route towards one tenant's subnet would name two next hops. Invariant 2 therefore carries two requirements, and violating either breaks something different: the identifier at the hypervisor and the subnet in the routing.

A machine of a tenant has one interface, on the tenant's overlay. The virtual router has two, one on the overlay and one on the management network, and it is instantiated from a template of the platform. Traffic leaving the tenant crosses one object per tenant because of the way the interfaces are laid out, and nothing has to be kept in step as machines are created and destroyed.

The management network is protected by its absence from every tenant's allowlist. It belongs to the platform administrator, no group is granted anything on it, and a tenant asking to attach a machine to it does not match any rule.

5.2.3 Compute and placement

The quota written for a tenant's group is evaluated when a machine is allocated, and its counters move as machines are created and destroyed, so a tenant that has reached its limit reads a refusal at allocation time.

Two independent filters narrow the set of hosts a machine may land on. The scheduler picks from what survives both. The first filter is the allowlist of the tenant's VDC names identifiers, which is what makes a host permitted. The second filter is the requirement expression of Section 4.5.1, which names properties, with `VM_MAD` for the driver and `TEE_TYPE` for the kind of enclave a host offers. If a machine's requirement is not satisfied by any host, it stays pending, and a host outside the allowlist is refused whatever the requirement says.

5.2.4 Storage

Each tenant has an image datastore of its own, so what one tenant stores never appears in another tenant's accounting. The datastore carries an explicit limit on the space it may occupy, 50 GB by default and raised per tenant from the tenant's own entry, which bounds the share of the shared backend one tenant can take. The backend under those datastores is one filesystem.

The base image a tenant boots from is cloned into that datastore instead of referencing the platform's catalogue. On this path no tenant holds a right on another tenant's image, and a datastore emptied at decommissioning takes with it only objects of that tenant, leaving untouched the catalogue every other tenant's clone was taken from. The price is one copy of the base image per tenant, which we accepted for this work because the alternative is a shared object whose permission field would have to grant USE to every tenant group at once.

What stays shared at this level is the datastore in which the disks of running machines live, since a machine's disk is created there when it is deployed. Section 5.5

says what are the limitations of this approach.

5.2.5 Where a cross-tenant call stops

The disjoint overlays invariant has no refusal behind it. Distinct identifiers and disjoint subnets are established when the entry is written into the tenant database, by the operation that allocates them and refuses a value another tenant already holds. Nothing at the front-end compares one tenant's overlay with another's, so an entry edited by hand crosses no gate that would catch a duplicate.

The other four are sustained by the platform itself. Figure 5.2 places them on the path a cross-tenant call takes, with one call each. Quotas and placement do not appear there, because a tenant exhausting its own quota is not calling on another tenant's objects. Table 5.2 instead pairs each invariant of Section 5.1.2 with the mechanism that sustains it.

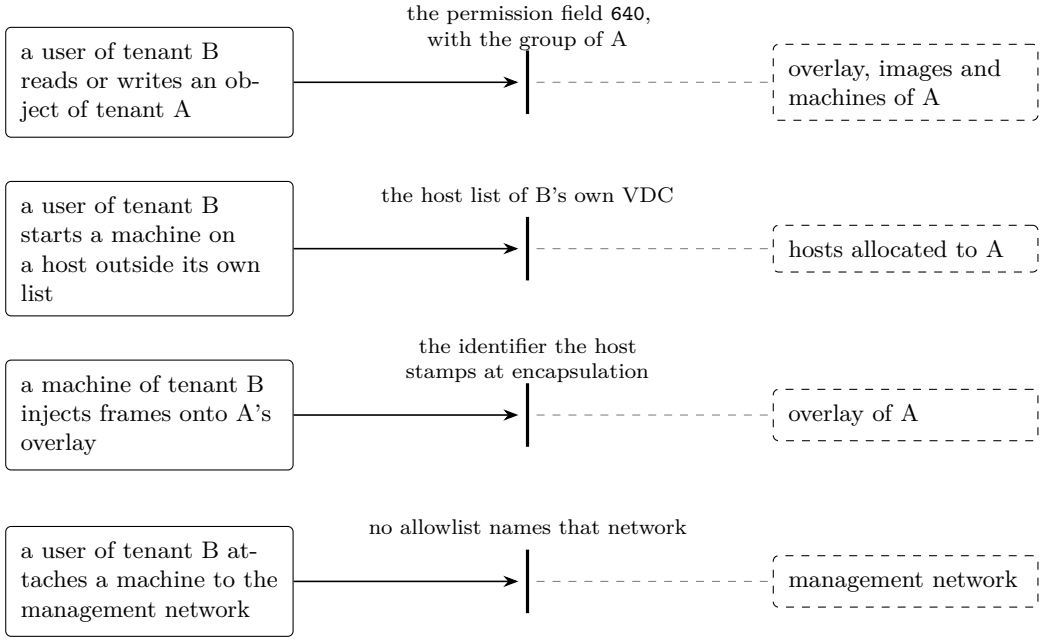


Figure 5.2: Four cross-tenant calls, the mechanism that refuses each of them, and the object each one never reaches.

5.3 Threat Analysis

In 1999, Microsoft created a framework called STRIDE that is used to analyze the security of a system [46]. The word itself is an acronym for the six categories of threats: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. We applied STRIDE to the model of Section 5.1, taking the six categories against each boundary that model draws. The analysis is organised by layer (hypervisor and authorisation, storage, network) applying the framework to each layer. Each threat carries an identifier, composed of a letter followed by

Table 5.2: Each invariant, the mechanism that sustains it, and what happens to a call that would break it.

Invariant	**Mechanism**	**What the offending call meets**
Attribution	Permissions `640` with the tenant's group on the overlay, the security group, the images and the machines. The router carries the administrator's group, and the datastore and the VDC carry no permission field	No rule matches the caller, and the front-end refuses the call
Disjoint overlays	One VNI per tenant, stamped by the host at encapsulation. The values themselves are allocated when the entry is written, with no check at the front-end	The frame is delivered to no other overlay. A duplicate written by hand meets nothing
Closed allowlist	Per-tenant VDC naming hosts, the platform image datastore and the tenant's own overlay, one identifier at a time. The tenant's group belongs to no other VDC	The resource is outside every rule the caller's group holds
One crossing	The management network appears in no allowlist. Only the virtual router holds an interface on it	The attachment matches no rule, and no machine of the tenant is on that segment to begin with
Bounded consumption	Group quota over processors, memory and machines, evaluated at allocation and raised only with the ADMIN right over the group	The allocation is refused when the counters are at the limit

a number. The letter is the category of the threat, and the number is the threat number within the category.

5.3.1 Adversaries and what is trusted

Four adversaries appear in the tables that follow, all of them inside the perimeter of one tenant:

- **A1**: a user of a tenant who has only read access to the tenant's objects;
- **A2**: a user of a tenant who looks for a way into another tenant's objects;
- **A3**: the group admin of a tenant, who creates and removes the users of that group;
- **A4**: a machine of a tenant whose operating system has been taken over, so that the adversary holds root inside the guest.

A4 is how the first exclusion of Section 5.1.2 comes back into the analysis. The model does not say what happens inside a guest, and that operating system is the customer's to administer, so what the platform separates has to stay separated while one guest is hostile.

In this model, five things are treated as trusted:

- the control plane, composed of the front-end daemon, the scheduler and the database behind them;

- the hypervisor on the nodes, at the patch level installed there;
- the account the framework applies with, together with the workstation that holds the Terraform state, the variable files and the access keys, which is reachable only through the corporate VPN;
- the secrets held by the continuous integration runner;
- the platform operator.

Figure 5.3 places the four adversaries and the trust boundary on the topology they act on. Three adversaries are left outside: a hostile platform operator, an adversary with physical access to the underlay, and the compromise of the supply chain that the OpenNebula image and the Terraform provider come from. The natural language interface is outside these tables for a reason that belongs to the model rather than to the scope of the analysis. A tenant has no control plane of its own and reaches the platform only through the front-end API under the rights its group carries, so none of A1 to A4 stands on the path that runs from a sentence to a plan. That path is operated by the cloud provider, and what bounds it is the sequence where a request becomes a plan, the plan is judged, and an operator approves it.

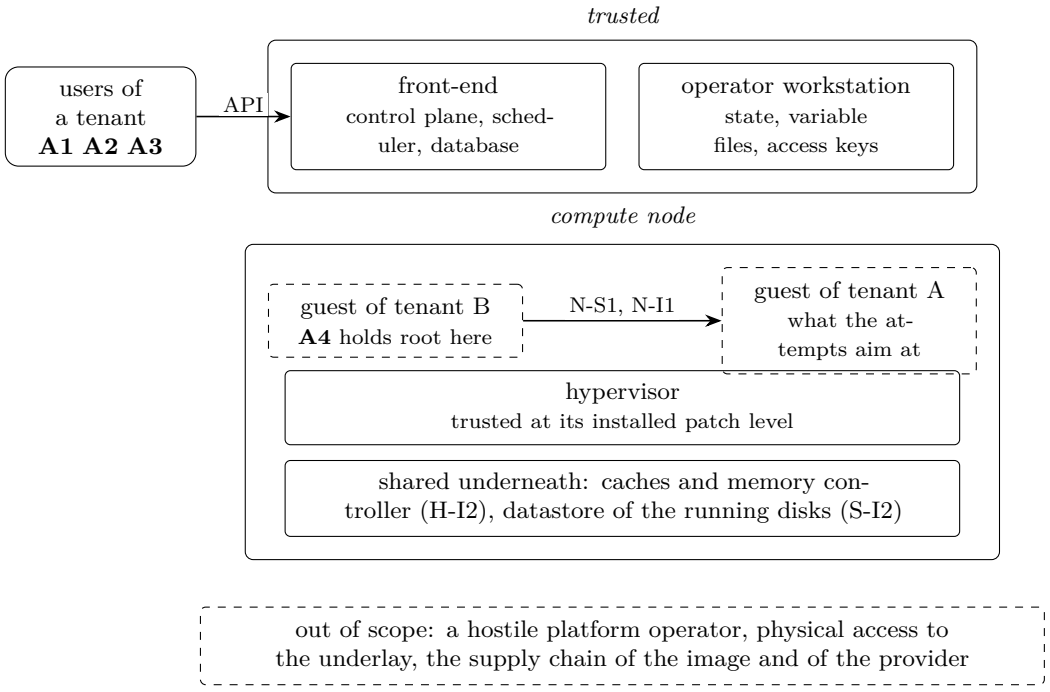


Figure 5.3: Where each adversary sits, what the analysis treats as trusted, and what it leaves outside.

5.3.2 Hypervisor and authorisation layer

The boundary of this layer runs around the compute hosts, the lifecycle of the machines placed on them, and the authorisation that decides who may run what and where. Identity sits here because the group, the VDC and the group admin enter the

analysis through the calls and the placements they admit. Table 5.3 collects the ten threats of this layer.

Table 5.3: Threats at the hypervisor and authorisation layer.

Threat	What the adversary tries	What it meets
H-S1 Spoofing	A2 acts as a user of another tenant, with credentials it obtained	Every call carries the identifier of a user the front-end authenticated. Custody of the credentials is the customer's, and no mechanism of the platform stands in for it
H-T1 Tampering	A2 edits a machine or an image of another tenant	The permission field `640` with the owning tenant's group. The caller falls in the third class
H-R1 Repudiation	A2 denies a destructive action on a shared host	The front-end records the call with the user identifier and a timestamp. The record stays on the front-end
H-I1 Disclosure	A2 lists the machines of another tenant through the API	The VDC bounds what the caller resolves, and the permission field denies the object itself
H-I2 Disclosure	A4 reads the state of a co-resident machine through a microarchitectural side channel	Nothing, for an ordinary workload. The mechanisms of Section 5.2 act on calls, and this attack issues none
H-D1 Denial of service	A2 takes shared compute capacity until none is left	The group quota over processors, memory and machines, counted when a machine is allocated
H-D2 Denial of service	A4 keeps the processors or the disk queue of a host busy, slowing a co-resident machine	Nothing. The quota bounds what a tenant may hold, not what it does with it
H-E1 Elevation	A3 moves from group admin to platform administrator	The rights of a group admin end at the users and the objects of its own group, and the VDC is not one of them
H-E2 Elevation	A4 escapes the guest onto the host through a hypervisor defect	Outside the boundary. The hypervisor is trusted at its installed patch level
H-E3 Elevation	A2 starts a machine on a host outside its own list	The host list of the tenant's VDC, and, for a machine asking for an enclave, the requirement expression as a second filter

The group admin holds more rights than any other adversary inside the model, and what bounds A3 is what a tenant never receives. It creates and removes the users of its own group and manages the objects of that group, and neither reaches outside the group. The quota written for the group and the VDC that carries the allowlist were deliberately kept away from it, because a tenant able to edit either would set the bounds it is then judged against. For the VDC the provider offers no way to hand the object to a tenant at all, since it exposes neither a group nor a permission field on that resource.

The quota is counted when a machine is allocated, which says what a tenant may hold and says nothing about what it does with what it holds. A machine that keeps a processor busy or fills the disk queue slows a machine of another tenant on the same host, and no rule refuses it. Bounding consumption rather than allocation would take per-machine processor pinning and a bandwidth policy on the block devices of each node.

H-I2 is the one threat of this layer that reaches across the boundary the hypervisor draws without issuing a call. Machines of two tenants on one host share its caches, its

memory controller and its branch predictors, and the attacks in that family read state through that sharing. SGX is the technology the platform exposes against it. An enclave holds its memory encrypted and outside the reach of privileged software, and its threat model assumes that the operating system and the hypervisor may both be hostile [47]. The platform gives a tenant the capability. A host labelled for it carries a real EPC, the compute module writes a region of 512 MB into every machine of a tenant that declared `sgx`, and inside such a machine we ran an unmodified workload in a genuine enclave, whose measurement at runtime equalled the measurement produced when the workload was signed. The workload is placed in an enclave only if the tenant puts it there, so an ordinary workload meets H-I2 exactly as it did before.

However, the chain that delivers the enclave memory is nested three times, from the physical host to the main machine, to the compute node, and to the machine of the tenant, and that platform is registered with no attestation service. So, an SGX quote generated on it stops at the retrieval of the platform certificates. A tenant therefore cannot prove to a third party which host its enclave runs on. In the context of this thesis TDX is not used, so it is a label and nothing more. A machine asking for it stays pending, because no host in the installation carries that label.

The record behind H-R1 is written by the front-end, which logs every authenticated call with the identifier of the user who issued it. That is what makes an action attributable once it has happened. The file stays on the front-end, where whoever administers the platform can edit it, and this analysis treats that operator as trusted.

5.3.3 Network layer

The boundary of this layer runs around the tenant overlays, the virtual routers, the security groups and the shared management network. Table 5.4 collects its eleven threats.

The virtual router is the object of a tenant an attacker gains most from, because it is the one object that touches both the tenant's overlay and the shared segment, and whoever configures it decides where that tenant's traffic goes. That is why it carries the group of the platform administrator rather than the group of the tenant it serves. The same reasoning covers the way into it from the network side, since the private half of the key that opens a session on the router stays on the operator workstation and only the public half is written into the guest at contextualisation. A tenant does not log into its own router either.

N-I1 depends on where the encapsulation happens. A guest chooses what it puts in a frame, and the host stamps the identifier and strips it again on the way back, so a guest that puts its interface in promiscuous mode reads the frames of its own segment and nothing else. N-R1 rests on the same header from the other direction, since a frame captured on the underlay carries the identifier of the overlay it came from and therefore names its tenant.

The security group of a tenant is declared with its rules, all outbound traffic and, inbound, connections to the secure shell port and ICMP. Version 1.5.0 of the provider

Table 5.4: Threats at the network layer.

Threat	What the adversary tries	What it meets
N-S1 Spoofing	A4 forges frames onto the overlay of another tenant	The identifier the host stamps at encapsulation. A frame carrying another one is delivered nowhere
N-S2 Spoofing	A4 answers address resolution or address assignment on the management network	No machine of a tenant holds an interface on that network
N-T1 Tampering	A2 edits the overlay of another tenant, its address ranges or its security group	The permission field `640` with the owning tenant's group
N-T2 Tampering	A4 takes over the virtual router of another tenant to rewrite where its traffic goes	The router carries the administrator's group, and the private half of the access key never enters a guest
N-R1 Repudiation	A4 denies having originated a flow that left the platform	The encapsulation header carries the identifier of the overlay, so a captured frame names the tenant that emitted it
N-I1 Disclosure	A4 reads the overlay traffic of another tenant from inside its own machine	Encapsulation and decapsulation run on the host. A guest is bound to one identifier and sees no other
N-I2 Disclosure	A4 reads the traffic of the management network	Only the virtual routers hold an interface on that segment
N-D1 Denial of service	A4 floods the management segment and denies the uplink to other tenants	The whole traffic of a tenant crosses one interface of its own router. No rate limit is configured on that interface
N-D2 Denial of service	A tenant exhausts the space of overlay identifiers	Identifiers are allocated when an entry is written into the tenant database, which no tenant reaches
N-E1 Elevation	A2 attaches a machine to the management network	That network appears in no tenant's allowlist
N-E2 Elevation	A2 adds its own segment to the virtual router of another tenant	The router is not the tenant's to modify, and the permission field denies the caller

fails when the security group attachment of an existing network is updated, so the object is created, tracked in state, and left unbound, and a rule set reaches traffic only if it is attached to a machine or to an interface by hand. The consequence stays inside one tenant, where a compromised machine reaches every other machine of the same tenant with nothing to narrow it. At this layer, what separates two tenants is the segmentation.

5.3.4 Storage layer

The boundary of this layer runs around the image datastores, the images themselves and the calls of the image API. The block traffic of a running machine on its host belongs to the hypervisor layer and is treated there. Table 5.5 collects the nine threats of this layer.

Table 5.5: Threats at the storage layer.

Threat	What the adversary tries	What it meets
S-S1 Spoofing	A2 clones the base image of another tenant into a machine of its own	The clone requires read on the source image, which the permission field denies
S-T1 Tampering	A2 writes into the base image of another tenant	Write belongs to the owner, and the owner is the account the framework applies with
S-T2 Tampering	The disk of a running machine is altered on the host	Outside the boundary. It rests on the integrity of the host and of the storage backend
S-R1 Repudiation	A2 denies which image it deployed	The clone, the persistence flag and the binding of an image to a machine are recorded with the image and user identifiers
S-I1 Disclosure	A2 reads a disk of another tenant through the image API	The datastore of the other tenant appears in no allowlist, and its images carry the permission field
S-I2 Disclosure	A4 reads what another tenant left in the shared datastore of the running disks	Nothing at the level of the object model. That datastore is one for the installation
S-D1 Denial of service	A2 fills the datastore of another tenant	Each tenant has an image datastore of its own, with its own limit
S-E1 Elevation	A2 has its contextualisation script executed on the host	Contextualisation runs inside the guest, and the channel does not reach the userspace of the host
S-E2 Elevation	A2 attaches an image of another tenant to its own machine and reads it from inside	The attach requires read on the image, which the permission field denies

S-I2 is the threat this layer leaves standing. The disk of a running machine is created in the datastore the whole installation shares, so the disks of two tenants are two files under one directory tree on the nodes. What separates them there is the ownership of a file and the permission field on the image object. Every other row of this table rests on a boundary of the datastore, and at this level there is none. Closing it takes a system datastore per tenant and volumes encrypted with keys of the tenant's own, and the second of the two is a manual procedure in the lifecycle rather than an object the framework declares.

The limit that answers S-D1 is a number the front-end compares against what the

tenant has already registered. No partition on the backend stands behind it. Each tenant's datastore carries its own limit, so a tenant that fills its own share leaves the accounting of the others untouched, while the blocks it consumed came from the one filesystem underneath all of them. A tenant is bounded in what it may register, and the space that remains for the others is bounded by what everyone registered together.

The contextualisation channel is the one path along which a value written in the declaration ends up executing somewhere. It executes inside the guest, run by the package the image carries at first boot, and the host takes no part beyond writing the values into the machine's template. A tenant that controls what goes into contextualisation controls its own guest, which it already administers.

5.3.5 Residual risks

The analysis leaves some of the threats standing, gathered here so that the limits of the model are readable in one place:

- **H-I2**: microarchitectural side channels between machines on one host, for every workload the tenant does not place in an enclave;
- **H-D2**: the contention a co-resident machine causes, which the quota does not bound;
- **H-E2**: an escape from a guest onto its host, which rests on the patch level of the hypervisor;
- **H-R1**: the record of the calls, which stays on the front-end and is not forwarded to a collector outside the reach of whoever writes it;
- **N-D1**: the absence of a rate limit on the interface each tenant crosses to reach the outside.

An installation with real attestation and a registered platform would close the first one for enclave-resident code.

5.4 Security Guarantees

Each property below is stated with the mechanism that sustains it and with the assumption it rests on. Four of them are refused by the platform while it serves a call, two are sustained by the procedure around an apply and hold while that procedure is followed, and the last describes a capability offered to a tenant rather than a protection applied on its behalf.

1. **Confinement of the objects.** No user of a tenant reads, writes or uses an object of another tenant through the front-end API. Three things hold it together: the permission field `640` with the tenant's group on the four kinds of object that carry one, the VDC that names hosts, datastores and networks

one identifier at a time, and the tenant's group belonging to no other VDC. It assumes that every object of a tenant was created by the framework. An object created beside it carries whatever the creator gave it.

2. **Separation of the segments.** A frame emitted by a machine of one tenant is not delivered on the overlay of another, because the host stamps the tenant's identifier at encapsulation and the guest never touches that field. The uniqueness of the identifiers comes from the operation that writes an entry into the tenant database, which rejects a value another tenant already holds, while nothing at the front-end compares one tenant's overlay with another's.

3. **One crossing towards the management network.** Machines owned by a tenant's group do not hold an interface on the shared segment. The only machine of a tenant that does is the instance of its virtual router, which carries the administrator's group. It assumes that the tenant does not obtain the private half of the key that opens a session on that router, which stays on the operator workstation.

4. **Bounded allocation.** No allocation takes a tenant's group over the quota written for that group, and the group cannot raise it. The counters move as machines are created and destroyed, and a tenant at its limit reads a refusal when it asks for one more. The property covers processors, memory and the number of machines at the moment of allocation, and says nothing about how hard a machine works the host it landed on.

5. **No change without a judged and approved plan.** Every change to a tenant reaches the front-end as a Terraform plan that passed the policy set and an explicit approval, and the path the automation takes contains no branch around either. The policy rules carry their own unit tests, and the continuous integration pipeline evaluates the whole set over the databases at every change to a policy or to a database.

6. **Attribution and revocation.** A tenant is ten kinds of object, each carrying either the tenant's name or its group, so closing a contract is an enumeration over a known set. The audit bundle carries the entity dumps, the lines of the front-end log that name the tenant and the slice of Terraform state that describes it.

7. **Enclave capability.** On a host labelled for it, a machine of a tenant receives a real EPC region and can create and run enclaves. In this way, inspection of a workload is impossible to perform by an external party.

5.5 Limitations

What is checked without an operator is narrower than the model. The pipeline that evaluates the policy set holds no plan job, because the state of each root is a local

file and the roots below read the state of the roots above them, which not holded by a runner. A difference between the declaration and what the front-end actually holds therefore surfaces when an operator runs a plan, and at no fixed interval.

The following objects carry less than their declaration suggests:

- the zone declared for the default VDC is empty and the provider does not converge on an empty list, so no plan withdraws a grant that reappears on that object. What isolates a tenant from it is the empty allowlist left after the import. OpenNebula adds every new group to that VDC and the provider cannot remove those entries, so a restored stock grant is cleared only by a call at the CLI;
- the image datastore and the VDC carry neither a group nor a permission field, because the provider exposes neither attribute on those two resources. For them the refusal rests on the allowlists alone, with no second mechanism behind the first.

Every property of Section 5.4 assumes that the framework is the only writer of the objects it declares. An object created next to the declaration, from a session opened at the CLI, carries the group and the permission field its creator gave it, and nothing later corrects it. The framework itself applies as a member of the group that holds every grant in OpenNebula, since a narrower identity could create neither a group, nor a VDC, nor a datastore. So, the identity that builds a tenant is the identity that reads and destroys the objects of every other tenant. The state of the four roots, the variable files and the private half of the access key sit together on one workstation reachable through the corporate VPN, which we treat as trusted.

The attestation gate answers a narrower question than its shape suggests. For each tenant that declared a TEE the gate builds a document with the fields of an SGX quote, signs it with a key pair the operator generated once, and admits the apply only if five checks hold, which are the signature over the canonical report body, the requested type against the header, the requested type against the label the host carries, a security version of at least one, and a timestamp inside a window of 300 s. Three of the five run on real values, since the signature and the freshness are computed as they would be against hardware, and the label is read from the front-end with `onehost show` while the quote is built. The enclave measurement, the processor security version and the certificate chain are mocked.

A verdict that passes therefore says that the record the front-end holds for that host still carried the requested label at the moment of the apply, and that the document was signed by the key the platform holds. Removing the label from the host makes verification fail, which is the property we wanted from the gate. It says nothing about the host itself, because a hardware verdict rests on certificates issued by the manufacturer's provisioning service for a registered platform.

Chapter 6

Implementation

6.1 Integrating Terraform and OpenNebula

6.1.1 The provider and its credentials

The version of Terraform that this work uses is constrained from 1.5.0 upwards, which is the release that introduced the `import` block used below. The lock file written by `init` records the checksum of the plugin binary, so the same configuration resolves to the same plugin on another machine.

The provider reaches `oned` over XML-RPC, and four root variables carry what the connection needs: the endpoint, a username, a password, and a flag that skips certificate verification. The last three are marked `sensitive`, so their values are withheld from the output of a plan. Locally they come from `terraform.tfvars`, a file excluded from version control, and in the pipeline they arrive as `TF_VAR_opennebula_*` environment variables read from the repository secrets. The endpoint answers only from inside the corporate VPN, so an apply issued from outside that perimeter fails at the first refresh.

Each root keeps its state in a local file beside its own configuration. That choice is what makes the dependency between two roots a relative path. The `terraform_remote_state` data source of `02_tenant_management` points at `../01_platform/terraform.tfstate` and reads the outputs of the platform root straight from the working copy. The four state files of the four roots therefore live on the workstation from which the apply is issued, and the repository carries only the configuration and the three databases.

6.1.2 From tenant primitives to resource types

A tenant is sixteen resources before it holds a machine of its own, and Table 6.1 names which type of the provider realises each part of the model described in Chapter 4 and Chapter 5. The overlay carries two address ranges because the address the virtual router answers on has to be reserved before the router claims it, and a range of fifty addresses starting at the hundredth host of the subnet is what the tenant's own machines lease from. The router is three resources, since the object, the machine

that runs it and each of its two interfaces are declared separately.

Table 6.1: The resource types that realise a tenant, and the module each one is declared in.

Part of the model	Resource type	Module
Group and its accounts	`opennebula_group`, `opennebula_user`	`identity`
Group administrators	`opennebula_group_admins`	`identity`
Quotas	`opennebula_group_quotas`	`identity`
Overlay	`opennebula_virtual_network`	`networking`
Addresses on the overlay	`opennebula_virtual_network_address_range` ×2	`networking`
Filtering rules	`opennebula_security_group`	`networking`
Router	`opennebula_virtual_router`, `..._instance`, `..._nic` ×2	`networking`
Image datastore	`opennebula_datastore`	`storage`
Base image of the tenant	`opennebula_image`	`storage`
Confinement	`opennebula_virtual_data_center`	`tenant_onboarding`
Machines	`opennebula_virtual_machine`	`compute`

The VDC is declared in the composition module and not in `identity`, where the group it binds is created. Its zone names the overlay by identifier, and that identifier is an attribute of a resource the networking module produces, so the reference crosses two sibling modules and can only be written where both are called. Declaring it beside the group would mean passing the number in by hand, and the dependency that orders the graph would disappear with it. Listing 6.1 shows the whole of the resource declaration.

Listing 6.1: The per-tenant VDC, in `modules/tenant_onboarding`. The three lists are the only resources the group's access control rules resolve to.

```
resource "opennebula_virtual_data_center" "this" {
  name       = "${var.tenant_name}-vdc"
  group_ids = [tonumber(module.identity.group_id)]

  zones {
    id             = 0
    host_ids       = var.host_ids
    datastore_ids = [var.datastore_id]
    vnet_ids       = [tonumber(module.networking.vnet_id)]
  }
}
```

6.2 Ansible Automation

6.2.1 The lab and the deployment chain

The OpenNebula project publishes a collection of playbooks and roles to install an own platform. It is called **one-deploy** [48], and it is distributed as the collection `opennebula.deploy`. This work uses it to install a custom platform from scratch.

What belongs to this thesis is the inventory that describes the installation, the script that builds the machine the platform schedules onto, and the three additions described in Section 6.2.2.

The installation is nested, and a single physical machine runs the whole cluster. On it, one virtual machine holds the OpenNebula front-end and acts as a hypervisor in turn, and inside that machine a second one is the compute node the platform schedules onto. Figure 6.1 shows the three levels and where a tenant's machines end up. The inner machine is created by `create_nested_vm.sh`, a script that writes a qcow2 disk backed by the Ubuntu Noble cloud image, generates a cloud-init user description and a network description with a fixed address and the operator's public key, and boots the result with `virt-install --import`. The flag that matters is `--cpu host-passthrough`, which exposes the instruction set of the physical processor to the guest. Without it the SGX features of the processor stop at the outer machine and the compute node has nothing to pass to a tenant.

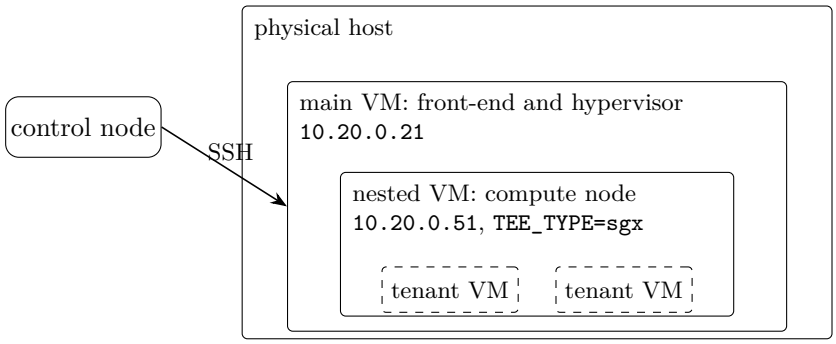


Figure 6.1: The three levels of the lab installation. The compute node is itself a guest of the machine that runs the front-end.

The entry point is `playbooks/main.yml`, which imports three playbooks in order: a local fix described in Section 6.2.2, then `opennebula.deploy.pre`, then `opennebula.deploy.site`. The first of the two upstream playbooks brings the machines to the state the installation assumes, covering the Python interpreter the modules need, the name resolution between front-end and node, the keys, time synchronisation, filesystems, kernel parameters and the binding of PCI devices to their drivers. The second installs the platform in three stages, each reachable on its own through a tag. The first stage builds the front-end, from the database to `oned` and the web interface, and pauses in the middle to elect a leader and let the handlers restart the daemon before the rest proceeds. The second stage installs the hypervisor, the datastore and the network drivers on the node. The third returns to the front-end for what could not be configured until a node existed. Being able to rerun one stage makes a failure recoverable without rebuilding the cluster.

The inventory in Listing 6.2 describes this installation using mocked data. It pins OpenNebula 7.0, selects SSH as the transfer mode of the datastores, and turns off Ceph, EVPN and Prometheus, none of which the lab runs. The administration network is declared there as well, on bridge `br0` over the physical interface `enp1s0`, with a range of 48 addresses. The node is reached through the front-end with

`ProxyJump`, because it has no route from outside the physical host, which is a consequence of the nesting. Its host variable `tee_type` is where a property of the hardware enters the deployment.

Listing 6.2: Extract from `inventory/cluster.yml`. Addresses follow the anonymisation used throughout this thesis.

```
all:
  vars:
    one_version: '7.0'
    features: {prometheus: false, ceph: false, evpn: false}
    ds:
      mode: ssh
    vn:
      admin_net:
        template:
          VN_MAD: bridge
          PHYDEV: enp1s0
          BRIDGE: br0
          AR: {TYPE: IP4, IP: 10.20.0.100, SIZE: 48}
frontend:
  hosts:
    fe-01: {ansible_host: 10.20.0.21}
node:
  vars:
    node_hv: kvm
    ansible_ssh_common_args: '-o ProxyJump=ubuntu@10.20.0.21'
  hosts:
    n1:
      ansible_host: 10.20.0.51
      tee_type: sgx
```

6.2.2 Additions to the collection

For this installation, there were three things that were written that the collection does not do.

- `fix-sources.yml` rewrites the Ubuntu package sources on every host so that they are fetched over `https` rather than `http`. On the network the lab sits on, a transparent proxy serves stale content over the unencrypted transport, the package manager reports a hash mismatch, and every role that installs anything fails with it. The playbook is the first import of `main.yml` because everything after it installs packages. Its single task uses `ansible.builtin.replace`, which reads the file before writing and therefore reports no change on a second run, and it carries `failed_when: false` so a host whose sources live elsewhere is passed over instead of stopping the run.

- `teardown.yml` removes an installation and returns both machines to the state they were in before it. The collection ships no such playbook, and rebuilding the lab by hand between experiments was the alternative. The play that empties the node runs before the play that empties the front-end, because the node is reached through the front-end and dismantling the front-end first would cut the

only route to it. Every task carries `failed_when: false`, because a teardown is most often asked for after a deployment that stopped part way through, and a service that was never installed must not stop the removal of the ones that were.

- `gui/fireedge_custom` replaces the web interface installed by the upstream `gui` role with a build including a customized graphical user interface. The build runs on the control node, takes some minutes because of the dependency install, and is copied to a staging directory on the front-end before being put in place. The command used to copy the build is `rsync -rlD --no-perms --no-owner --no-group`.

The chain ends with the two hosts registered on the front-end, ready to be used.

6.3 Code and Module Organization

6.3.1 Languages and repositories

There are main directories that hold the code used in this work.

- `/IaC` carries the Terraform configuration, the policy set and the scripts that carry out an operation.
- `/one-deploy` carries the installation of Section 6.2.1, which is an inventory and three additions to a published collection.
- `/slm-interface` carries the natural language interface described in Section 6.4.

Table 6.2 shows what each language is used for and how many lines of code there are in this project. They are counted over the source files with comments and blank lines included.

Table 6.2: The languages of the three repositories, and the size of each part.

Language	What it holds	Lines
HCL	six modules and four state roots	2 558
Python	the interface, its tools and the chat interface	2 054
Python	the tests of the interface and the model benchmark	2 309
Python	the attestation scripts	578
Bash	the operations, their gates and the script that builds the nested machine	940
Rego	four policies and their unit tests	482
YAML	the inventory and the additions to `one-deploy`	287
YAML	the continuous integration workflow	145
JSON	the three databases, with one tenant declared	33

Each language is confined to one kind of statement:

- HCL declares the objects that have a lifecycle in the front-end's database and an identifier other objects refer to;

- YAML, read by Ansible, describes the state of a host, which is a package installed and a file written;
- Rego states what must not be true of a proposed change;
- Bash puts in sequence programs that already exist, such as `terraform`, `opa`, `jq` and `ssh`, and computes the filters that assemble a document;
- Python is where a schema and a hierarchy of classes carry the work, which is the interface and the attestation scripts.

6.3.2 Modules and their interfaces

A module under `modules/` declares resources and doesn't hold any state, and a root under `environments/` holds the state and reads the files. There are six modules which are called from the four roots, with one exception. `tenant_onboarding` is a composition module, and it calls `identity`, `networking` and `storage` and hands the identifier each one returns to the next, which is how the group created first becomes the owner of the network and of the datastore created after it.

The interface of every module has the same shape. It receives the tenant name and the identifiers of objects created elsewhere (the management network, the router template and its image, the cluster and the datastore). It returns identifiers, one for each object the module created. The reading of `tenants.json` happens once, in the root, in a `locals` block that decodes the file, and the `for_each` over the decoded map turns one key of that file into the sixteen resources of Table 6.1.

Names are derived from the tenant name with a fixed suffix, so `tenant-a` owns `tenant-a-grp`, `tenant-a-vdc`, `tenant-a-vxlan-vnet`, `tenant-a-sg` and `tenant-a-vrouter`, so the list of objects on the front-end says whose each one is without a lookup in the state. Two values are not derived from that name. The bridge of the overlay is `vxbr-` followed by the VNI, because the kernel takes fifteen characters for an interface name and a longer name would be truncated. The addresses of the machines come from `cidrhost` over the subnet the tenant was given, with the address range holding fifty addresses, and the machine of index i with starting address X taking host $X + i$.

There are two modules, `compute` and `vm`, that both create virtual machines. `compute` builds the machines of a tenant, and the values it needs arrive from the layer above, which are the group, the network, the subnet and the image cloned into the tenant's datastore, so its shape is fixed and its entries differ only in number, processors and memory. `vm` builds a machine on infrastructure that already exists and resolves those same values from its own variables, so it takes an image by identifier, by URL or by clone. One module for both would carry two sets of variables of which one is always unused, and a condition on which set was filled. The cost of the separation is the parts they state twice, which is the context block, the disk and the attachment to a network.

6.3.3 Provider and packaging limitations

Two attributes of the template needed to expose real TEE and GPU hardware to a machine have no field in version 1.5.0 of the provider. The first is the PCI vector that assigns the GPU to a machine, for which neither the template nor the machine resource has a block. The second is `VALIDATE="NO"` inside the `RAW` section that carries the enclave configuration. The SGX memory backend is declared in a custom `qemu:` namespace, which libvirt refuses unless the validation of that section is turned off, and the provider's own `raw` block writes a type and a payload with no field for it. Both are written through `template_section`, a generic block that emits a named vector of key-value pairs into the body of the template, as in Listing 6.3. The output of the two vectors was read back from the front-end with `one.template.info` after the apply.

Listing 6.3: The two `template_section` blocks in `environments/01_platform`.

```
template_section {
  name = "PCI"
  elements = {
    VENDOR = var.gpu_vendor_id
    DEVICE = var.gpu_device_id
    CLASS  = var.gpu_class_id
  }
}

template_section {
  name = "RAW"
  elements = {
    TYPE     = "kvm"
    VALIDATE = "NO"
    DATA     = local.gpu_sgx_raw_data
  }
}
```

The same escape hatch appears in `modules/vm`, inside a `dynamic` block whose iterator is empty unless the entry declares an enclave size, so an ordinary machine does not carry any `RAW` section.

The second gap is in the packages a compute node installs. OpenNebula registers a host per driver, so a node that runs both hypervisors appears twice in the front-end's database, and the packaging makes that impossible on one machine. `opennebula-node-kvm` and `opennebula-node-lxc` have a conflict issue in their Debian and Ubuntu packaging, so only one of the two can be installed on a single machine. Containers would therefore take a second compute node. The consequences are that `workloads.json` carries `num_lxc` equal to zero in every entry, the container resource in `modules/compute` is declared with `count = var.lxc_count` and produces nothing, and the shared container image and template stay declared in `01_platform`, where they cost one image in a datastore. To turn containers back on it is needed a second node registered with the LXC driver and a number changed in a database. There modules can be unchanged.

6.3.4 The operations as shell scripts

There are seven scripts under `scripts/` that carry out the operations from a terminal, and every one of them sources a library of 64 lines. The library holds the paths of the three databases and of the roots, the helpers that print and exit, and `require_tool`. `require_tool` stops the run at its first line when `jq` or `opa` is missing, so if there is an error it comes up at the beginning of the launch. Two important functions of the script are `confirm_destructive` and `ssh_frontend`.

`confirm_destructive` prints what is about to be removed, reads a line from the terminal, and exits unless what was typed is the tenant key. `ssh_frontend` runs a command on the front-end as `oneadmin`, and encodes the command in base64 before sending it and decodes it on the other side. Written plainly, a parenthesis or a quote in the caller's argument closes the wrong quoting level and the command arrives truncated.

The order of the gates is written in the script. `deploy_workloads.sh` calls `opa_check.sh` first, then generates and verifies a quote for every tenant whose `tee_type` is not `none`, and only then reaches `terraform apply`. `offboard_tenant.sh` exports the audit bundle before it touches any database, then applies layer 03 and layer 02 in this order.

Every script takes `--dry-run`, used to run the gates and report without blocking and where `terraform plan` replaces `terraform apply`. That form is what the interface calls to produce the plan an operator reads. The offboarding dry run restores both databases before it returns, since it has to remove an entry in order to plan its removal.

6.3.5 Policies and continuous integration

The policy set is composed by four files of Rego, for a total of 210 lines, with 272 lines of unit tests beside them and three files holding the values for the rules. Every rule is a `deny` rule in the same package, so the whole set produces one set of messages. A run that passed is a run that produced the empty set. Listing 6.4 is one example of rule on quotas.

Listing 6.4: Example rule of `opa/policies/quotas.rego`. The message names the tenant, what was asked and the limit it crossed.

```
deny[msg] if {
    some tenant_key
    tenant := input.tenants[tenant_key]
    workload := input.workloads[tenant_key]
    requested := workload.kvm_cpu * workload.num_kvm
    requested > tenant.quota_cpu
    msg := sprintf(
        "%v: cpu quota exceeded (requested %v, limit %v)",
        [tenant_key, requested, tenant.quota_cpu],
    )
}
```

`opa_check.sh` assembles the document those rules read with `jq`, out of the three databases and the three data files, substituting an empty object for a database that is absent and a null for a data file that is absent. The gate is global, since each rule narrows itself to the part of the document it cares about. The verdict comes back from `opa eval` as a set, which OPA serialises as an object whose keys are the messages, so the script reads the keys and counts them. A record is written with a timestamp in `opa/opa_audit/` carrying the layer, the outcome and the messages.

The pipeline runs on **GitHub Actions** for development purposes, but it can be ran on any other provider that supports YAML formatting. It has three jobs:

- `lint` runs `terraform fmt -check` over the four roots.
- `policy` runs `opa check`, then the unit tests, then an evaluation of the whole set over the databases as they stand in the repository.
- `validate` runs `terraform validate` on each root in a matrix, after an initialisation with `-backend=false` that resolves the provider without reaching a state backend.

The version of Terraform is pinned to 1.5.7 in the workflow, and a job is triggered by a change to a `.tf` file, to one of the three databases, or to a policy or its data.

There is no plan job, because a root downstream reads the state of the root above it through a `terraform_remote_state` data source, and that file lives on the workstation the applies are issued from, not in the runner. A plan for `02_tenant_management` in the pipeline would fail on a missing file, so the plan stays a local step and the pipeline checks what can be checked without state.

6.4 Natural Language Interface Implementation

6.4.1 Three processes on one machine

For this work, the interface is composed of three processes, running all on the front-end machine, with 16 cores and 32 GB of memory. Ollama answers on port 11434, the backend runs under `uvicorn` on port 8000, and the chat interface, written with Streamlit, on port 8501. The generation of the tokens by the model runs on the processor, since no accelerator is attached to that machine.

The operations reach the infrastructure by writing files in the `IaC` repository and by running `terraform`, `opa` and `jq` as subprocesses, so the interface has to sit where those files and those programs are. Figure 6.2 shows the three processes and what the layer below them touches.

One file configures the whole model, and it carries the tag of the model, the address of Ollama, the address of the backend, the root of the `IaC` repository and the path of the database that holds the conversations. The model in use is `qwen2.5:3b`, and it is easily changable by switching the name of that tag. Chapter 7 reports what the models scored and its impact on the work.

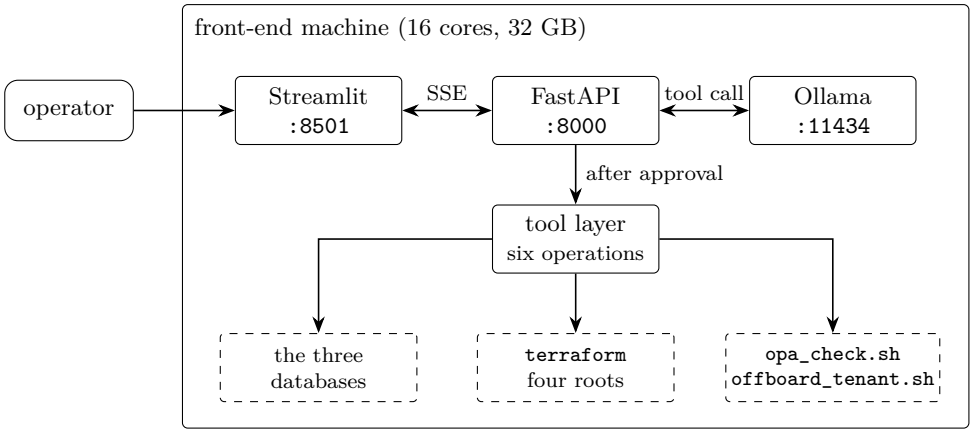


Figure 6.2: The three processes of the interface, and what the tool layer reaches below them.

6.4.2 The tool layer

In the code each operation is a Python class. `IacTool` is the abstract base that the six of them extend, and is composed by: a name and a description, a schema for the arguments, a `dry_run` that computes what the change would do, and an `execute` that takes the arguments and a boolean.

The base class refuses a subclass whose name or description is empty when the module is imported. The description travels to the model inside the schema of the tool and it is what the model reads in order to choose, so a tool without one is a tool that never gets picked and gives no reason for it. The same base class holds `_opa_check`, which runs `opa_check.sh` with a timeout of 30 seconds and turns a non-zero exit into an error carrying the output of the script.

The arguments of each operation are a Pydantic model and their field descriptions are written in Italian and travel with the schema, which makes them part of what the model is told. Onboarding declares one required field, which is the name of the tenant, and every other value has a default or is derived.

Derivation is an important step because it is where most of a tenant comes from. The tool reads the tenant database, takes the first VNI not in use starting at 1020, and computes the subnet, the gateway and the management address of the router from it. A value the request named explicitly is checked against the same database and it is refused when another tenant holds it.

6.4.3 Making the output of a small model usable

The system prompt is in Italian, as is everything the operator reads and types. Listing 6.5 shows an extract of a prompt used in this work. Its first rule tells the model to emit the call instead of describing it and forbids it from asking for a confirmation, because the confirmation is a step of the system and a model that asks for one in prose stops the conversation. Its second rule is the conversion of memory, with a handful of examples, to guide the model into choosing a correct parameter.

A prompt is not enough, and the code that surrounds the model exists to support

Listing 6.5: Extract from the system prompt. It is written in Italian, like the rest of the conversation.

```
Quando hai capito quale strumento serve e hai i parametri
obbligatori, CHIAMA SEMPRE lo strumento (emetti la tool call).
NON descrivere lo strumento a parole, NON scrivere la chiamata
come testo e NON chiedere conferma all'operatore: la conferma
e' gestita automaticamente dal sistema DOPO la tua chiamata.

REGOLA RAM. I campi di memoria (kvm_memory_gb, quota_ram_gb)
sono SEMPRE in GB. Se l'operatore fornisce la RAM in MB,
convertila in GB DIVIDENDO per 1024 prima di riempire il campo.
Esempi (input operatore -> argomento):
- "8192 MB di RAM" -> kvm_memory_gb: 8    (8192 / 1024)
- "512 MB per VM"  -> kvm_memory_gb: 0.5  (512 / 1024)
```

the small models in respecting the schema. A quota of zero is read as an argument the model invented, and a figure of 16384 arriving in a field measured in gigabytes is read as the operator's megabytes and divided, since nothing in this installation has sixteen terabytes of memory and the policy set caps a machine at 32 GB. Converting is better than refusing because the converted value will appear in the plan the operator reads before anything is applied, while a refusal would return an error for a request that was legitimate.

Three parameters of the model are set in code:

- Temperature is zero and the seed is fixed, which together make two runs of the same request comparable.
- Chain-of-thought output is turned off, because the model would stream its reasoning where the interface expects a structure.
- The context window is left at whatever the model declares, as raising it cost far more time per request without improving what came back.

6.4.4 The backend, the plan and the approval

The backend of the AI chatbot is built with a Python lightweight framework called FastAPI. Its two main endpoints return a stream of server-sent events. `/chat` emits the text of the reply token by token, and then one `plan_ready` event carrying the name of the tool, the arguments and the text of the plan. `/confirm` emits the output of the operation line by line and ends with a verdict. Three smaller endpoints answer what the interface needs in order to draw itself, which are the state of the backend, the tags Ollama has available, and the list of past conversations.

Between the two calls the plan waits in memory, in a dictionary holding one entry per session, and a session holds one plan at a time. `/confirm` removes it from that dictionary, instantiates its arguments against the schema a second time, and calls the operation with the approval set. A refusal writes a line in the history and returns, and a new message on the same session drops the pending plan before the model is called again.

The operations block the flow, since they run `terraform` and then wait, so they are called in a worker thread and the event loop stays free to stream. Offboarding is the exception that reads better as a stream. It starts the script as a subprocess and forwards its lines as they arrive, which is what puts the audit export and the two destroys in front of the operator while they happen. The conversations are kept in SQLite, in two tables. The title of each new conversation is produced by the model with a second call, in a background task.

The chat interface is a simple Streamlit web application named NextGPilot. It renders the plan in a block of its own with two buttons under it, and it disables the input box while a plan is waiting, so a session cannot stack a second request on top of a decision that has not been taken. On approval, it sends the identifier of the session and a boolean, and afterwards it shows what the stream returns, including the error and its details when a gate refuses.

The interface carries its own tests, which are 1 633 lines over three suites, covering the tools, the endpoints and the scenarios in natural language. The tool layer is mocked in all of them, so no test reaches Terraform or the front-end. Chapter 7 uses the third suite, which measures what the models get right on a corpus of requests.

Chapter 7

Evaluation and Results

7.1 Test Methodology and Experimental Setup

The numbers in this chapter come from three campaigns, all run on the installation of Section 6.2.1. Two of them measure the system that was built, and every figure they produce is a mean over repetitions written with its standard deviation beside it, in the form $\mu \pm \sigma$. The third is the reference point for the manual procedure, and it is an estimate described in the next section.

7.1.1 The measurement environment

Every measurement was taken on the OpenNebula front-end, called `fe-01` here, an Intel processor with 16 vCPU and 32 GB of memory, working on Ubuntu OS. Terraform, OPA, the Ollama server, the agent and the three databases all run on that one machine, and the compute node where the platform schedules the workloads is a virtual machine inside it. The cluster is nested twice, so a tenant's machine is a third-level guest and the absolute times below carry that nesting. A cluster on bare metal would produce different ones. Table 7.1 pins the version of every piece of software the timings ran against.

Table 7.1: Software the measurements ran against.

Component	Version	Role in the measurements
OpenNebula	7.0	the platform reached by an apply
Terraform	1.9.8	the binary the timings measure
OpenNebula provider	1.5	translates the declaration into XML-RPC calls
OPA	1.17.0	evaluates the policy set
Ollama	0.30.10	serves the four models, temperature 0.0
Python	3.12.3	runs the agent, the tools and the harnesses

The model campaign takes thirty minutes of continuous inference. Host names and tenant names are pseudonyms, kept consistent with the rest of this thesis. Software versions, hardware sizing and every measured time are the real ones used.

7.1.2 What the metrics mean

Four quantities recur in the rest of the chapter:

- **Step time** is the wall-clock duration of one step of an operation, from the moment its process starts to the moment it exits.
- **Tool accuracy** is the fraction of the scenarios of a corpus on which the agent selects the same operation as the reference.
- **Argument score** is the fraction of the reference argument keys that the call fills with the reference value, averaged over the scenarios where the operation chosen is the right one. It separates picking the wrong tool from picking the right tool badly.
- **Latency** is the time from the request to the last event of the reply stream.

The model campaign runs one pytest session per model, discards a warm-up run and unloads the model from Ollama before the next one starts. We adopted that protocol after finding that the result files produced during development were not comparable with each other. On the same model, gemma4-12b, a run with one session per model reports a mean latency of 5.56 s, and a run that put five models in a single session reports 54.9 s. Ollama loads and evicts weights between one scenario and the next, and the load time ends up inside the measured latency. Table 7.2 gives the size of the two campaigns those quantities come from.

Table 7.2: The two measured campaigns behind this chapter.

Campaign	What it produces	Size
Lifecycle timings	the duration of every step of onboarding, audit export and offboarding	5 complete cycles, and 3 more for the decomposition of the offboarding
Model comparison	tool accuracy, argument score and latency of four models on one corpus	4 models over 29 scenarios, 3 repetitions, 1 for mistral-7b

The manual procedure has no campaign behind it. Typical values of the time needed for the manual operation are nowadays in the range of one hour for creating a single VM and two hours for a full tenant. That reference point has no repetitions and no dispersion, and it covers the written procedure in full. It is written as an estimate wherever it appears, without a standard deviation.

7.2 Tenant Lifecycle: Automated vs. Manual Process

The test pipeline that was built was timed on a complete cycle, from the policy gate that precedes the first apply to the removal of the tenant and its record. Five repetitions ran without a failed step.

7.2.1 The cost of one lifecycle

The tenant that was timed is the simplest and smallest one the operation builds: a single KVM machine with one processor and 1024 MB of memory, `tee_type` set to `none`, on a VXLAN segment of its own. Each repetition reproduces the sequence the onboarding operation executes, which is the policy gate, the two entries written into the databases, the apply of layer 02 and then the apply of layer 03. The plan of each root was timed before its apply, since that is the cost the operator waits between the request and the approval. The tenant and workload databases are saved before the campaign and restored after it. Table 7.3 reports each step over the five repetitions.

Table 7.3: Duration in seconds of each step, over five complete cycles.

Step	mean	median	st. dev.	min	max
policy gate	0.17	0.17	0.00	0.16	0.17
write of the two entries	<0.01	—	—	—	—
`plan` layer 02	0.92	0.92	0.02	0.89	0.93
`apply` layer 02	26.92	25.09	4.03	22.16	31.18
`plan` layer 03	0.77	0.77	0.03	0.73	0.80
`apply` layer 03	28.36	30.36	4.50	20.31	30.39
onboarding, end to end	**57.13**	57.33	6.75	47.25	63.38
audit export	7.59	7.60	0.25	7.24	7.92
offboarding ceremony	232.68	254.97	47.53	164.90	271.16

Onboarding a tenant takes 57.1 s ± 6.8. The two applies account for 55.3 s of it and everything else together for less than two seconds. The apply of layer 02 creates sixteen OpenNebula objects, while the apply of layer 03 creates one object, the machine. The two costs are nonetheless comparable, 26.9 s against 28.4 s, which is the first sign that in OpenNebula the price of an operation follows the state transitions it waits for rather than the number of objects it touches.

A plan costs less than a second on either root. In the flow that asks for an approval the operator's wait is therefore set by the model. The audit export costs 7.6 s ± 0.3, and the offboarding runs it before it touches anything, so a tenant cannot be removed without the record of what it was.

Figure 7.1 puts the two operations on one axis, with the same fill marking the same layer in either direction. The sixteen objects of layer 02 cost 26.9 s to create and 56.0 s to remove, while the single machine of layer 03 goes from 28.4 s to 155.6 s.

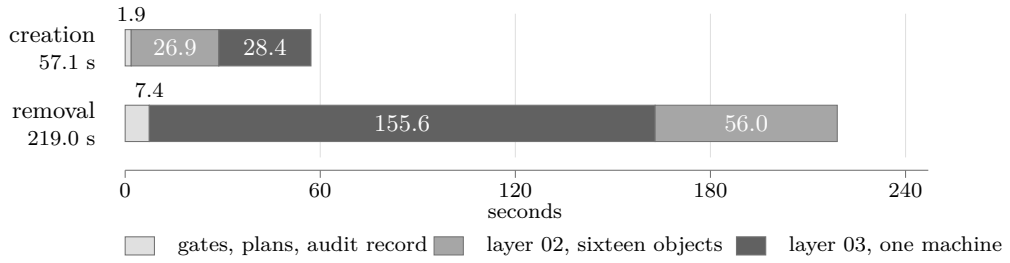


Figure 7.1: Where the time of one lifecycle goes. The removal bar is the decomposition of the offboarding ceremony, measured over three repetitions.

7.2.2 Destruction costs four times creation

The five removals took 232.7 s ± 47.5. Every step of creation has a standard deviation between 0.02 and 4.5 s, while the removals spread from 164.9 to 271.2 s. Figure 7.2 shows the ten runs one by one, with a band of one standard deviation on either side of each mean. The band is 13.5 s wide for the creations and 95.1 s for the removals.

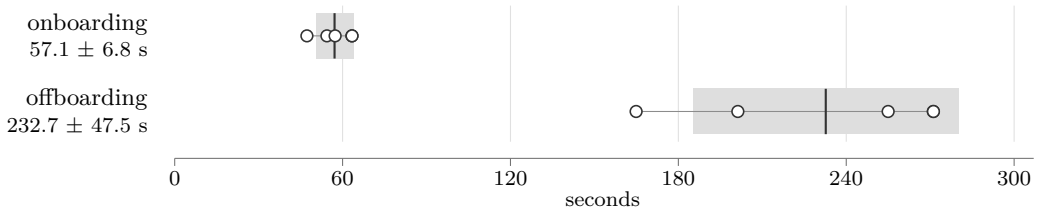


Figure 7.2: The five repetitions. Each dot is one run, the vertical rule is the mean and the band is one standard deviation on either side of it.

Creating a resource returns as soon as the resource is defined, while destroying one waits on state transitions OpenNebula drives asynchronously, and a machine has to be shut down and have its disk removed before the datastore under it can go. This is why the cost does not scale with the number of objects: layer 02 destroys sixteen of them in a third of the time layer 03 needs for one.

7.2.3 Comparison with the manual process

The comparison between the result of 57.1 s for an onboarding with the manual process time of two hours is misleading. The estimate covers the written procedure in full, including the paperwork that is its heavier part, while the measured figure covers the provisioning actions alone. The precise measurement of how much of the two hours is operation and how much is documentation is not known. These numbers provide an upper bound on what automation can take out of an activation. Table 7.4 puts the two side by side.

Table 7.4: The manual procedure and the automated pipeline.

	Manual, as estimated	**Automated, as measured**
A new tenant, end to end	about two hours, procedure and paperwork together	57.1 s ± 6.8, provisioning only
A single machine	about one hour, inside an existing tenant	28.4 s ± 4.5, the apply of layer 03 as the closest equivalent
Evidence of what was done	four documents, compiled by hand after the fact	one of the four documents, 7.6 s ± 0.3, generated from what the platform holds
Actions the operator takes	several tens of commands in a fixed order	two entries in two files, and two applies

A manual activation produces four documents: the entry in the activation register, the entry in the key register, the hardening checklist and the pre-production verification report. The automated pipeline produces the equivalent of one. The audit bundle costs 7.6 s, it is assembled from what the platform already holds, and

the offboarding produces it before it destroys anything. The other three documents stay manual for this work.

7.3 Comparison of different models

Four models were measured on one corpus, served by Ollama on the machine used for the tests, which has no accelerator. What is scored is the choice of one operation among the five that change a tenant: onboarding, scaling, quota update, offboarding and the deployment of a single machine.

7.3.1 The corpus and the candidates

The corpus holds 29 requests written in Italian in seven categories. Five categories name an operation and expect the call that performs it, the other two ask for something else. `discrimination` phrases a request so that it could be read as either of two nearby operations. `ambiguity` withholds a value the operation requires, and the answer that counts as correct is an **abstention**, a question back to the operator with no call attached.

Table 7.5 lists the five candidates. All but one are quantised to Q4_K_M, and the memory column is the size of the weights once loaded, which on a front-end that also runs Terraform, the databases and the three processes of the interface is the figure that decides whether a model can stay resident.

Table 7.5: The candidate models, largest first.

Model	Ollama tag	Param.	Quant.	Memory
Gemma 4 12B	`gemma4:12b`	11.9 B	Q4_K_M	9.0 GB
Mistral 7B Instruct v0.3	`mistral:7b-instruct-v0.3-q4_K_M`	7 B	Q4_K_M	4.4 GB
Phi-3-mini	`phi3:mini`	3.8 B	Q4_0	2.2 GB
Llama 3.2 3B	`llama3.2:3b`	3 B	Q4_K_M	2.0 GB
Qwen 2.5 3B	`qwen2.5:3b`	3 B	Q4_K_M	1.9 GB

`phi3:mini` never reached the corpus. Every request to it returns HTTP 400 with `model does not support tools`, before the prompt is processed, because the Ollama Modelfile for that model carries no `{{ .Tools }}` section in its template and Ollama has nowhere to inject the schema of the operations. The later packaging, `phi3.5:3.8b`, behaves the same way.

Temperature is zero in every run reported here. Before fixing it we swept the nine values from 0.0 to 0.8 in steps of 0.1 on the two 3B models, and the pass/fail distribution does not move: the same scenarios fail at every value. What a model gets wrong is anchored in the model rather than in the sampling, so fixing the temperature doesn't reduce accuracy and makes a run repeatable.

7.3.2 Accuracy, arguments and latency

Table 7.6 reports the four chosen models over the whole corpus. gemma4-12b is the most accurate at 96.6%, ten points above llama3.2-3b and seventeen above qwen2.5-3b. It also fills arguments best, with a score of 0.991. It pays for both in latency, 5.56 s ± 1.40 against about 2.3 s for either 3B model, which is what its larger parameter count costs on processors alone. The model mistral-7b was only measured against 1 repetition, because what it produces is a binary negative rather than a score worth averaging. It emits no call at all on the operating scenarios of the corpus, so the other runs went to the three models that can be compared with each other.

Table 7.6: Four models over 29 scenarios. Latency is per request, over 87 requests for the models with three repetitions. The argument score of mistral-7b is computed over its three abstentions, which have no arguments to fill.

Model	rep.	accuracy	arg score	latency	p95
gemma4-12b	3	96.6% (28/29)	0.991	5.56 ± 1.40	8.01
llama3.2-3b	3	86.2% (25/29)	0.959	2.30 ± 0.58	3.49
qwen2.5-3b	3	79.3% (23/29)	0.977	2.28 ± 0.77	3.72
mistral-7b	1	10.3% (3/29)	1.000	2.94 ± 0.72	3.99

7.3.3 Three ways of failing

Table 7.7 divides the corpus by category and Figure 7.3 by what went wrong. No model ever selected the wrong operation on a request that names one, and the two `discrimination` scenarios, written to be read the wrong way, were answered correctly by every eligible model. Each error in the campaign is either an argument the model failed to supply or an action taken where the correct answer was a question.

Table 7.7: Scenarios answered correctly by category, for one repetition. The three repetitions coincide.

Category	scenarios	gemma	llama	qwen	mistral
onboard	4	4	4	3	0
scale	3	3	3	2	0
quota update	5	5	4	4	0
offboard	4	4	4	4	0
deploy machine	8	8	8	7	0
discrimination	2	2	2	2	0
abstention	3	2	0	1	3
Total	**29**	**28**	**25**	**23**	**3**

gemma4-12b loses a single scenario out of 29. The one it loses asks to add a tenant without naming it, and the model calls `onboard_tenant` instead of asking for the name. Its six operating categories are perfect, arguments included.

llama3.2-3b is perfect on the operating categories except one `update_quota` that leaves out `tenant_name`, and it fails all three abstentions. It means that if it is faced with a request it cannot fill, it acts.

qwen2.5-3b loses six scenarios, and four of them are the same mistake in four places, which is a required field left out (`tenant_name` in `scale_tenant` and in `update_quota`, `vm_name` in `deploy_vm`) or an explicit `null` in `quota_ram_gb`. The other two scenarios lost are on abstentions, making qwen better than llama and worse than gemma.

mistral-7b emits no call at all in the 26 operating scenarios. It answers in prose, describing what it would do and listing the parameters, which is the behaviour the system prompt was written to suppress. The conclusion is that the model is not able to perform any tool call and is therefore not useful for this work.

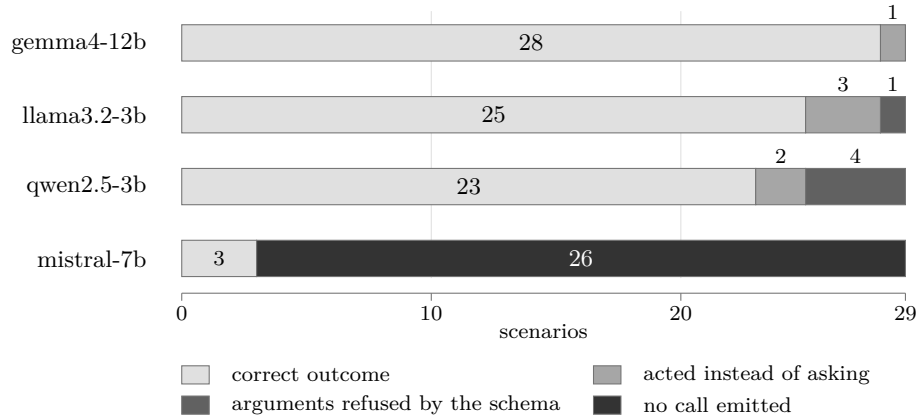


Figure 7.3: How the 29 scenarios of one repetition end, for each model.

7.4 Discussion

The absolute figures of this chapter refer to the machine they were taken on. A cluster nested twice stretches every state transition OpenNebula waits for. The same operations on bare metal would surely produce smaller numbers. The important part to catch in this analysis is the shape of the results, which is independent of the hardware. Destruction costs about four times creation and a plan costs about a thirtieth of the apply behind it.

The different latencies of the models depend mostly on the environment. Ollama serves them on processors, which is where the factor of 2.4 between gemma4-12b and a 3B model comes from.

The corpus is 29 requests with one phrasing each, and it was written alongside the tools it exercises. The descriptions the model reads inside the schema and the sentences it is scored on come from the same hand, which is a favourable condition for the interface.

The manual estimate is not a measurement. It is one value reported by one person, without repetitions and without a spread, and it covers a procedure wider than the one the pipeline automates. A part of the comparison that is even is the audit bundle, which replaces one of the four documents and is assembled from what the platform already holds.

Two limits of the timing campaign are worth naming. Every repetition provisioned one tenant at a time. It is not tested what the same pipeline does when several onboardings are requested together. The tenant that was timed is also the smallest the operation builds (one machine with one processor and 1024 MB), so the chapter reports the cost of a lifecycle at that size.

Chapter 8

Conclusions and Future Work

8.1 Summary of the Work

The starting point of this work was to consider a generic system where a tenant was the result of a sequence of actions carried out on the OpenNebula front-end and of the documents written around them. The written procedure fixed the order of those actions and not the result, so two activations that both respected it could differ in a quota, in a permission bit, or in the cluster a datastore was attached to. The constraints that mattered for security were sentences in a document, checked by the person executing the procedure, and the register that made the activation auditable was compiled by hand out of information the platform already held. An estimation of that manual work is one hour for a single machine and about two hours for a full tenant.

The first stage of this work expresses the sequence of that manual work as Terraform. Six modules describe identity, networking, storage and compute, and four state roots apply them. The four states are separate so that a change to one tenant is independent to the platform underneath it. Continuous integration checks the formatting of every root, runs the policy set with its unit tests, and validates each root after an initialisation that resolves the provider without reaching a state backend.

The second stage defines what separates one tenant from another and writes the definition of it. A tenant is one VDC with its own group, access control rules and quotas, and the host, datastore and network allowlists of that VDC are the only resources its rules resolve. Every object the tenant owns carries the permission field `640` and the tenant's group, its machines sit on a VXLAN segment of their own, and the only machine of the tenant holding an interface on the shared segment is its virtual router. The model was analysed with a STRIDE approach across three boundaries, and 210 lines of Rego with 272 lines of unit tests evaluate every planned change before it is applied. They cover data residency, image provenance, quotas and placement on a host labelled for a TEE.

The third stage puts a natural language interface in front of those operations. Six tools reach the framework, and a model served locally by Ollama chooses among

them and fills their arguments. No information leaves the premises, and no change reaches the front-end without a dry run, an evaluation of the policy set, and an operator who approved the plan those two produced.

Chapter 7 measured the results on the installation it was built on. Onboarding a tenant takes 57.1 s ± 6.8 against an estimate of about two hours for the procedure it replaces, and the audit bundle costs 7.6 s ± 0.3. On a corpus of 29 requests, the most accurate of the four models measured answered 28 correctly, and no model ever selected the wrong operation on a request that names one.

8.2 Assessment of the Approach

8.2.1 The infrastructure layer

Declaring a tenant in Terraform gave back what it was chosen for. Several tens of commands in a fixed order became two entries in two files and two applies, and all the evaluations of Chapter 7 show that they produced the same sixteen objects without a failed step. The declaration is also where a resource says why it exists, so deciding whether a resource still belongs to an active customer consists in reading an entry in a file, and no longer means inspecting the running state and reconstructing the intent behind it.

The resistance the approach met came from the provider. Three places in the declaration carry less than the model behind them:

- version 1.5 exposes no field for the PCI vector that assigns a GPU to a machine and for the flag that turns off validation of the `RAW` section carrying the enclave configuration. In this work, both are written through a generic block that emits a named vector of key-value pairs;
- the default VDC is imported with its zone and group lists under `ignore_changes`, because the provider does not converge on an empty list, and a stock grant that reappears on that object is cleared by a call at the CLI;
- the image datastore and the VDC do not carry a group or a permission field, because the provider exposes neither attribute on those two resources, so for them the refusal rests on the allowlists alone.

Each of the three is a documented workaround around a gap in the provider, so none of them required stepping outside Terraform to apply a resource by hand.

8.2.2 Checking before applying

The policy gate costs 0.17 s inside an onboarding of 57.1 s, 0.3% of the operation it guards. A sentence in a document, read by the person executing the procedure, can be a rule with unit tests beside it and a message that names the tenant, what was asked and the limit it crosses.

The attestation gate is exercised from one end to the other and verified in part. It builds a document with the fields of an SGX quote, signs it with a key pair the operator generated once, and admits the apply only if the signature holds over the canonical report body, the requested type matches the header, the requested type matches the label the host carries, the security version is at least one, and the timestamp falls inside a window of 300 s. The signature, the freshness and the label are computed on real values, while the enclave measurement, the processor security version and the certificate chain are produced by the harness. The pipeline has the shape a real attestation service would sit in, and it is missing the verdict on the hardware.

8.2.3 The natural language layer

The model used for the interface, `qwen2.5:3b`, answers 23 of the 29 scenarios of the corpus correctly, and none of its six errors reaches OpenNebula. Four of them leave a required field out or fill it with an explicit null, and the schema of the tool refuses the call before a plan exists. The other two are calls emitted where the correct answer was a question back to the operator, and those arrive in front of the operator as a plan to approve. The interface therefore stands on the checks around the model, at a size where the model alone would not carry it.

The corpus also says where the difficulty is. No model ever selected the wrong operation on a request that names one. Every error in the campaign is an argument the model failed to supply or an action taken where the correct answer was a question. Choosing among a restricted number operations is within reach of a 3B model served on processors, and perfectly filling their arguments and knowing when to stop is not.

The model in service was chosen for its latency. It answers in 2.28 s $\pm$ 0.77 against 5.56 s $\pm$ 1.40 for gemma4-12b, and it occupies 1.9 GB of resident weights against 9.0 GB on a front-end that also runs other heavy processes.

The corpus holds 29 requests with one phrasing each, and the descriptions the model reads inside the schema were written alongside the sentences it is scored on, which is a favourable condition for the interface. What the campaign establishes is that a model small enough to be served on a front-end can drive a fixed set of infrastructure operations, provided a schema, a policy set and an operator stand between it and the platform.

8.3 Future Work

8.3.1 Short-term Extensions

The three extensions below work on the code and the data that already exist. The platform already offers the hardware and the capability to support them.

- **Three documents are still written by hand.** The key register entry, the hardening checklist and the pre-production verification report name values

the platform already holds, such as which datastore was created, under which identifier, with which quota and which owner. The audit bundle shows the shape of the work, since it collects that material in 7.6 s from the entity dumps, the front-end log and the Terraform state. Extending the export script with a template for each of the three documents turns them into products of the run. This is the extension that could be carried out first.

- **A wider corpus.** The 29 requests of Chapter 7 carry one phrasing each. Several phrasings per intent, and a second author who has not read the schemas, would measure what the interface does with language it has not seen. The harness exists and one campaign over four models is thirty minutes of inference, so the cost of the extension is the writing of the new scenarios.

- **Retrieval augmentation over the repository.** What the model knows about the installation reaches it through the descriptions inside the schema of each tool and through the code that derives a tenant's network values from the tenant database. A retrieval layer over the repository would put the module interfaces, the policy set and the current contents of the three databases in front of the model at the moment it answers.

8.3.2 Long-term Extensions

The four extensions below need new hardware or a change in what the model is asked to do, differently from today.

Attestation against real hardware. Closing the gap of remote attestation means registering the platform with the manufacturer's provisioning service, obtaining the certificate chain and the collateral it issues, and verifying an enclave measurement produced by the enclave.

A second technology behind the label. A tenant that declares SGX receives a real EPC region on a host labelled for it, and a tenant that declares TDX receives a value the workload carries and a label the scheduler matches, with nothing behind it. Supporting it takes hosts that expose the technology, a path from the host to the guest for the trust domain, and a second evidence format in the attestation gate.

An identity narrower than the one that applies today. The framework applies as a member of the group that holds every grant in OpenNebula, because a narrower identity could create neither a group, nor a VDC, nor a datastore. The identity that builds a tenant is therefore the identity that can read and destroy the objects of every other tenant. Splitting the run into a privileged part that creates the container of a tenant and an unprivileged part that fills it would bound the reach of the second, and the split has to be argued against the primitives the platform offers.

A record of the calls outside the machine. The front-end log stays on the front-end, which the threat analysis leaves as a residual risk, and the audit bundle is assembled from that same log. An operator with administrative credentials can therefore edit the record of their own actions and the evidence built from it.

Forwarding each call to a collector the front-end can append to and not rewrite closes the risk, and the collector has to sit outside the machine the automation runs on.

8.3.3 Towards Production Deployment

The framework was built and measured on a laboratory cluster nested twice inside one physical host. One part of it has run against the production cluster, which is the standalone root: a single machine was deployed there onto infrastructure that already existed, through the same operation and the same policy gate the laboratory uses. The three roots that build a tenant have never been applied outside the laboratory.

An inventory of the production cluster taken during this work compared it against what the repository assumes, and the differences are structural:

- the management network differs from the one the platform root declares in name, in bridge and in address space, so an apply as it stands would create a second network beside the one in use instead of adopting it;
- image storage is backed by a distributed cluster filesystem, while the examples in the repository target datastores local to a node, so the datastore parameters and the behaviour of every image clone have to be set for that backend;
- part of the shared baseline already exists on the cluster, including the default VDC, a base cloud image and templates created by hand, and each of those objects has to be imported into the state, referenced by identifier, or left outside the declaration;
- the hosts publish their enclave capability under attributes of their own and not under the label the laboratory stamps and the scheduling requirement of a workload names, while the policy set and the wrapper script assume that every workload with a TEE lands on one host;
- the three databases in the repository hold laboratory records, and a production onboarding starts from empty ones.

The order of a possible future work follows from the list. The repository is made production-aware first, by parameterising or importing the shared objects that exist and by removing the dependency on the container image. The placement and attestation model is then adapted, by reading the attributes the hosts publish instead of the laboratory label and by dropping the assumption of a single host from the policy set and from the deployment script.

The documentation of an onboarding can be automated the same way the audit bundle already is, taking the three manual procedures inside the current framework.

Bibliography

[1] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. "A systematic mapping study of infrastructure as code research". In: *Information and Software Technology* 108 (2019), pp. 65–77. DOI: 10.1016/j.infsof.2018.12.004 (cit. on p. 7).

[2] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. "A Survey of DevOps Concepts and Challenges". In: *ACM Computing Surveys* 52.6, 127 (2020). DOI: 10.1145/3359981 (cit. on p. 7).

[3] Florian Beetz and Simon Harrer. "GitOps: The Evolution of DevOps?" In: *IEEE Software* 39.4 (2022), pp. 70–75. DOI: 10.1109/MS.2021.3119106 (cit. on p. 7).

[4] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, João Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. "What serverless computing is and should become: the next phase of cloud computing". In: *Communications of the ACM* 64.5 (2021), pp. 76–84. DOI: 10.1145/3406011 (cit. on p. 7).

[5] Michele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 580–589. DOI: 10.1109/ICSME.2019.00092 (cit. on pp. 7, 18).

[6] Akond Rahman, Chris Parnin, and Laurie Williams. "The Seven Sins: Security Smells in Infrastructure as Code Scripts". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 164–175. DOI: 10.1109/ICSE.2019.00033 (cit. on p. 8).

[7] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. "Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics". In: *IEEE Transactions on Software Engineering* 48.6 (2022), pp. 2086–2104. DOI: 10.1109/TSE.2021.3051492 (cit. on p. 8).

[8] Md Mahadi Hassan, John Salvador, Shubhra Kanti Karmaker Santu, and Akond Rahman. "State Reconciliation Defects in Infrastructure as Code". In: *Proceedings of the ACM on Software Engineering* 1.FSE, 83 (2024). DOI: 10.1145/3660790 (cit. on pp. 8, 20).

[9] Amazon Web Services. *What Is AWS Control Tower?* AWS Control Tower User Guide. URL: https://docs.aws.amazon.com/controltower/latest/userguide/what-is-control-tower.html (visited on 08/09/2026) (cit. on p. 9).

[10] Microsoft. *Subscription vending*. Microsoft Cloud Adoption Framework for Azure. URL: https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/design-area/subscription-vending (visited on 08/09/2026) (cit. on p. 9).

[11] Google. *Decide the resource hierarchy for your Google Cloud landing zone*. Google Cloud Architecture Center. URL: https://docs.cloud.google.com/architecture/landing-zones/decide-resource-hierarchy (visited on 08/16/2026) (cit. on p. 9).

[12] Google. *Landing zone design in Google Cloud*. Google Cloud Architecture Center. URL: https://docs.cloud.google.com/architecture/landing-zones (visited on 08/09/2026) (cit. on p. 9).

[13] Google. *Cloud Foundation Toolkit*. GoogleCloudPlatform on GitHub. URL: https://github.com/GoogleCloudPlatform/cloud-foundation-toolkit (visited on 08/09/2026) (cit. on p. 9).

[14] OpenStack Foundation. *Identity concepts*. Keystone documentation. URL: https://docs.openstack.org/keystone/latest/admin/identity-concepts.html (visited on 08/09/2026) (cit. on p. 9).

[15] OpenStack Foundation. *Adjutant documentation*. OpenStack. URL: https://docs.openstack.org/adjutant/latest/ (visited on 08/09/2026) (cit. on p. 10).

[16] Apache Software Foundation. *Roles, Accounts, Users, and Domains*. Apache CloudStack Administration Guide. URL: https://docs.cloudstack.apache.org/en/latest/adminguide/accounts.html (visited on 08/09/2026) (cit. on p. 10).

[17] OpenNebula Systems. *Multitenancy*. OpenNebula 7.0 documentation. URL: https://docs.opennebula.io/7.0/product/cloud_system_administration/multitenancy/index.html (visited on 08/09/2026) (cit. on p. 10).

[18] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. NIST Special Publication 800-162. Updated 2 August 2019. National Institute of Standards and Technology, 2014. DOI: 10.6028/NIST.SP.800-162 (cit. on p. 11).

[19] Open Policy Agent. *Open Policy Agent documentation*. Cloud Native Computing Foundation. URL: https://www.openpolicyagent.org/docs (visited on 08/09/2026) (cit. on p. 11).

[20] The Kubernetes Authors. *Admission Control in Kubernetes*. Kubernetes documentation. URL: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/ (visited on 08/09/2026) (cit. on p. 11).

[21] HashiCorp. *Manage policies and policy sets in HCP Terraform*. HashiCorp Developer. URL: https://developer.hashicorp.com/terraform/cloud-docs/workspaces/policy-enforcement/manage-policy-sets (visited on 08/16/2026) (cit. on p. 11).

[22] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. "Semantic-based Automated Reasoning for AWS Access Policies using SMT". In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8602994 (cit. on p. 11).

[23] Joseph W. Cutler et al. "Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (2024), pp. 670–697. DOI: 10.1145/3649835 (cit. on p. 12).

[24] Open Policy Agent. *Policy testing*. Cloud Native Computing Foundation. URL: https://www.openpolicyagent.org/docs/policy-testing (visited on 08/16/2026) (cit. on p. 12).

[25] Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava. *A Survey of using Large Language Models for Generating Infrastructure as Code*. Accepted at ICON 2023. 2024. arXiv: 2404.00227 [cs.SE]. URL: https://arxiv.org/abs/2404.00227 (visited on 08/09/2026) (cit. on p. 12).

[26] Patrick Tser Jern Kon et al. "IaC-Eval: A Code Generation Benchmark for Cloud Infrastructure-as-Code Programs". In: *Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track*. 2024. URL: https://proceedings.neurips.cc/paper_files/paper/2024/hash/f26b29298ae8acd94bd7e839688e329b-Abstract-Datasets_and_Benchmarks_Track.html (visited on 08/09/2026) (cit. on p. 12).

[27] Tianyi Zhang, Shidong Pan, Zejun Zhang, Zhenchang Xing, and Xiaoyu Sun. *Deployability-Centric Infrastructure-as-Code Generation: Fail, Learn, Refine, and Succeed through LLM-Empowered DevOps Simulation*. Accepted at FSE 2026. 2026. arXiv: 2506.05623. URL: https://arxiv.org/abs/2506.05623 (visited on 08/09/2026) (cit. on p. 12).

[28] Francesco Romeo, Luigi Arena, Francesco Blefari, Francesco Aurelio Pironti, Matteo Lupinacci, and Angelo Furfaro. "ARPaCCino: An Agentic-RAG for Policy as Code Compliance". In: *New Trends in Database and Information Systems*. Vol. 2676. Communications in Computer and Information Science. Cham: Springer Nature Switzerland, 2025, pp. 467–481. DOI: 10.1007/978-3-032-05727-3_39 (cit. on p. 13).

[29] Margaret-Anne Storey and Alexey Zagalsky. “Disrupting developer productivity one bot at a time”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 2016, pp. 928–931. DOI: 10.1145/2950290.2983989 (cit. on p. 13).

[30] Amazon Web Services. *What is Amazon Q Developer?* Amazon Q Developer User Guide. URL: https://docs.aws.amazon.com/amazonq/latest/qdeveloper-ug/what-is.html (visited on 08/09/2026) (cit. on p. 13).

[31] Microsoft. *Azure Copilot overview*. Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/copilot/overview (visited on 08/09/2026) (cit. on p. 13).

[32] Google. *Gemini for Google Cloud overview*. Google Cloud documentation. URL: https://docs.cloud.google.com/cloud-assist/overview (visited on 08/16/2026) (cit. on p. 13).

[33] Ansible Community. *ansible/ansible source repository*. Python; GNU General Public License v3.0 or later; sponsored by Red Hat. GitHub. URL: https://github.com/ansible/ansible (visited on 08/10/2026) (cit. on p. 16).

[34] Ansible Community. *Ansible concepts*. Ansible Community Documentation. URL: https://docs.ansible.com/projects/ansible/latest/getting_started/basic_concepts.html (visited on 08/10/2026) (cit. on pp. 16, 17).

[35] Ansible Community. *Introduction to Ansible*. Ansible Community Documentation. URL: https://docs.ansible.com/projects/ansible/latest/getting_started/introduction.html (visited on 08/10/2026) (cit. on p. 16).

[36] Ansible Community. *Glossary*. Ansible Community Documentation. URL: https://docs.ansible.com/projects/ansible/latest/reference_appendices/glossary.html (visited on 08/10/2026) (cit. on p. 17).

[37] HashiCorp. *Dependency Graph*. Terraform documentation. URL: https://developer.hashicorp.com/terraform/internals/graph (visited on 08/10/2026) (cit. on p. 19).

[38] OpenNebula Systems. *OpenNebula Terraform provider, version 1.5.0*. Terraform Registry. URL: https://registry.terraform.io/providers/OpenNebula/opennebula/1.5.0 (visited on 08/10/2026) (cit. on p. 19).

[39] OpenNebula Systems. *OpenNebula/one source repository*. GitHub. URL: https://github.com/OpenNebula/one (visited on 08/10/2026) (cit. on p. 22).

[40] OpenNebula Systems. *OpenNebula Overview*. OpenNebula 7.0 documentation. URL: https://docs.opennebula.io/7.0/getting_started/understand_opennebula/opennebula_concepts/opennebula_overview (visited on 08/10/2026) (cit. on p. 22).

[41] OpenNebula Systems. *Permissions and ACLs*. OpenNebula 7.0 documentation. URL: https://docs.opennebula.io/7.0/product/cloud_system_administration/multitenancy/chmod/ (visited on 08/16/2026) (cit. on p. 25).

[42] Ollama. *ollama/ollama source repository*. GitHub. URL: https://github.com/ollama/ollama (visited on 08/10/2026) (cit. on p. 26).

[43] Ollama. *Importing a Model*. Ollama documentation. URL: https://docs.ollama.com/import (visited on 08/10/2026) (cit. on p. 26).

[44] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. *Small Language Models are the Future of Agentic AI*. Preprint, not peer reviewed. 2025. arXiv: 2506.02153 [cs.AI]. URL: https://arxiv.org/abs/2506.02153 (visited on 08/10/2026) (cit. on p. 27).

[45] LangChain. *langchain-ai/langchain source repository*. GitHub. URL: https://github.com/langchain-ai/langchain (visited on 08/10/2026) (cit. on p. 28).

[46] Adam Shostack. *Threat Modeling: Designing for Security*. Indianapolis, IN: Wiley, 2014. ISBN: 978-1-118-80999-0 (cit. on p. 50).

[47] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. 2016. URL: https://eprint.iacr.org/2016/086 (visited on 08/12/2026) (cit. on p. 54).

[48] OpenNebula Systems. *one-deploy: OpenNebula Ansible playbooks*. GitHub. URL: https://github.com/OpenNebula/one-deploy (visited on 08/13/2026) (cit. on p. 61).