Ahmedabad University

Ahmedabad University

School of Engineering and Applied Science (SEAS)

Course Code: CSE540
Cloud Computing

Mini-Cloud Platform

Scheduler, VM Creation & Live Migration

Instructor
Professor Sanjay Chaudhary

Group Members

Name	**Roll No.**	**Email**
Shlok Shelat	AU2240025	shlok.s1@ahduni.edu.in
Shrey Salvi	AU2240033	shrey.s2@ahduni.edu.in
Purvansh Desai	AU2240036	purvansh.d@ahduni.edu.in
Dhananjay Khanjariya	AU2240023	dhananjay.k@ahduni.edu.in

November 17, 2025

Contents

List of Figures

1. Abstract

The rapid expansion of virtualization and cloud computing has created a growing need for lightweight, customizable, and cost-efficient private cloud platforms that educational institutions and small organizations can deploy without the complexity and overhead of full-scale commercial cloud systems. This project presents the design and implementation of a Mini-Cloud Platform built using XCP-NG, Xen-Orchestra (XOA), FastAPI, Celery, Redis, and PostgreSQL, enabling automated virtual machine provisioning, VM lifecycle management, asynchronous task execution, metrics collection, and VM migration through a unified controller.

The main goal of the project is to develop a full-fledged API-based, fully automated virtualization controller, which will be able to communicate with an XCP-NG pool to create, manage, and monitor virtual machines. The system is comprised of a RESTful API layer (FastAPI) and an asynchronous distributed task execution engine (Celery + Redis) and a relational database backend (PostgreSQL) to ensure reliable orchestration. The API and command-line features of Xen-Orchestra support VM tasks including template cloning, network attachment, power management, getting metrics and cross-host migrations.

Its architecture is a microservices-based design that is modular, consisting of a controller service, Celery worker cluster, Redis message broker, a PostgreSQL persistence layer, and the underlying XCP-NG infrastructure. The API server receives VM creation requests and forwards it to Celery workers which asynchronously executes provisioning tasks. A metrics-collection module is used for accessing system performance metrics (CPU, RAM, disk usage, network I/O) to XOA periodically, which are saved within the PostgreSQL database. The project also deploys a migration service through which a host-to-host live migration can be done seamlessly and using the task definitions of XOA.

The adoption of the Mini-Cloud Platform meant a stable and production-like environment with the ability to: provision VMs automatically using templates, process jobs asynchronously with stable and dependable task states, schedule metrics collection and storage, execute live VM migrations in structured API payloads and log and monitor all system components centrally.

The system is highly automated, and it is easy to use and extend. The main results are less manual intervention in the process of virtualization, standardized cloud-like behavior in the management of VM, and a scalable architecture, which may be extended to meet future improvements, like that of autoscaling, integration into a UI, or container act. The project shows that an academic value of open-source technologies allows to construct an effective lightweight private cloud environment and can be deployed into practical use.

2. Project Statement

2.1 Problem Definition

As organizations continue to embrace virtualization and cloud computing, there is need to have lightweight and customized private cloud solutions (especially in case of the academic institutions and small scale businesses). Nevertheless, the currently available enterprise cloud infrastructures including VMware vSphere, Proxmox VE, or OpenStack have numerous limitations: they are expensive, require a lot of time to install, rely on a wide range of services, and have steep learning curves. Likewise, as XCP-NG and Xen-Orchestra (XOA) provide a powerful virtualization stack, the experience of managing virtual machines (VMs) can be tedious and manual, where the administrators have to perform numerous tasks manually like:

- Create virtual machines using the dashboard from templates by hand,
- Observe monitoring metrics from various systems,
- Perform host-to-host migrations manually, (or maybe host machine-based),
- Check VM status by referencing log files or the XOA UI, and
- Complete provisioning tasks for individual machines.

When a number of virtual machines must be created, managed, and monitored concurrently, this is cumbersome. This frequently entails new delays, unpredictable configurations, and a greater likelihood of human error on the part of the system administration team.

As a consequence, there is a lack of centralized, automated, API-driven orchestration that is capable of integrating seamlessly with XCP-NG/XOA workflow covering VM lifecycle operations, metrics collection, task scheduling and migration without human action.

2.2 Objectives

The main goal of this project is to create and deploy a Mini-Cloud Platform that automates virtualization workflow processes within an XCP-NG environment using contemporary cloud-native tools and techniques. The objective of the system is to provide a fully automated controller that communicates with the virtualization layer through well-defined APIs and asynchronous background workers.

The key objectives are:

Technical Objectives

1. Create a FastAPI based rest controller to provide secure and modular API endpoints to provision, manage, monitor and migrate VMs.
2. Automate VM lifecycle actions, such as template creation, power, network connection, disk setups and destruction.
3. Utilize Celery in conjunction with Redis to run long-running functions in an asynchronous manner without blocking APIs.
4. Incorporate a persistence layer built on top of PostgreSQL to store metadata of VMs, logs of metrics, task requests and logs at the system level.
5. Install a Metrics Collection Module that periodically retrieves VM performance data off of XOA and then archives it to be analyzed.
6. Use VM Migration Service so that controlled and structured migrations can take place through XOA live migration APIs.
7. Design a microservices based architecture, which is scalable, modular, and enables the controller, workers, database, and the broker to scale independently.

Research and Learning Objectives

- Comprehend current cloud architectures and orchestration concepts.
- Obtain practical experience with virtualization platforms (i.e. XCP-NG, XOA).Gain hands-on experience with virtualization platforms (XCP-NG, XOA).
- Execute asynchronous programming, distributed task queues, and event-driven architecture.
- Analyze the pragmatics of cloud Infrastructure as a Service automation and DevOps methodologies.

2.3 Scope of the Project

The ultimate goal of the Mini-Cloud Platform is to develop an automated and reliable layer for managing virtualization. The scope of the project entails:

In-Scope Features

- VM creation with XCP-NG templates with API.
- Managing the lifecycle of VMs (Start, Stop, Reboot, Shutdown)
- Running long-running tasks in the background asynchronously.
- Database that tracks VM records and metrics that are centralized.
- Performance measures (CPU, RAM, Disk, Network) will be taken periodically
- XOA API live migration based on cross-host VM
- All the critical system operations should be logged and monitored
- The entire controller stack runs in a decoupled and dockerized format.

Technologies In Scope

- XCP-NG and Xen-Orchestra
- FastAPI
- Celery & Redis
- PostgreSQL
- Docker & Docker Compose
- Python 3.x

Out-of-Scope (Future Work)

The project has deliberately taken out advanced features based on time and resource limitations, as for example:

- Multi-tenant dashboard UI
- Automated autoscaling based on metrics
- Integration with Kubernetes
- Distributed storage provisioning
- User-level cloud billing system
- High-availability cluster for the controller

These can be added in future features.

2.4 Challenges Addressed

The project focused on mostly real-world issues with typical virtualization and cloud management:

1. Manual VM Provisioning

Multiple times with a lot of manual work, like cloning templates, connecting networks, and configuring resources is laborious. **Approach:** Fully automated VM creation workflow through FastAPI + Celery tasks.

2. Lack of Centralized Orchestration

XOA has a UI but does not provide a customizable orchestration layer for batching an operation or custom automation. **Approach:** A centralized API-controlled controller to manage VM life cycle, migrations, and metrics.

3. Synchronous Execution Limitations

Heavy operations like clones or migrations can't be run through normal APIs in a synchronous manner. **Approach:** Celery distributed workers with Redis message broker to execute asynchronous scalable processes.

4. Metrics Monitoring Complexity

To view the performance of a VM requires navigating through the UI of XOA manually. **Approach:** An automated scheduler collects and stores metrics periodically into a PostgreSQL database.

5. Migration Coordination

Live migrations are still manual to select the target host and storage repositories. **Approach:** Automated migration service using structured API payload and executing tasks through XOA.

6. Error Handling & Reliability

General CLI operations will not always provide meaningful logs instead failing silently without monitoring in a central location. **Approach:** Logging retries and structured exception handling inside Celery workers.

7. Scalability Limitations

As the number of VMs increases, manual processes become unmanageable. **Approach:** Horizontally scalable architecture with stateless controllers and distributed workers.

2.5 Motivation & Significance

Motivation

The modern IT infrastructure is supported by cloud computing. Nevertheless, a lot of institutions do not have the infrastructure or funding to implement full-scale implementations of private clouds.The need in this project was based on the need:

- A cheap and easy-to-use personal cloud service.
- Practical understanding and exposure to real-life virtualization technologies.
- Everyday administration is to be automated to minimize human error.
- Recreating hyperscale cloud behavior (provisioning, metrics, migration) with open-source software
- Writing a complete programmable interface on a virtualization platform.

Significance

The project has a great academic and practical significance:

- **Educational Importance:** It offers a full, real world implementation of the inner working of cloud provisioning systems.
- **Technological Impact:** Demonstrates how the open-source virtualization tools can be improved so that they act as a complete private cloud.
- **Operational Efficiency:** It lowers the administrative burden and fastens the VM deployments significantly.
- **Extensibility:** Architecture may be extended to allow autoscaling, dashboards, container support, etc.
- **Practical Deployment Use:** It can be used in research laboratories, academic institutions, as well as small data centers.

In this project, we show that a mini-cloud environment that is well structured, automated, based on API, and built with heavyweight cloud platforms is affordable.

3. System Overview

3.1 What is the Mini-Cloud Platform?

Mini-Cloud Platform is a small, modular and automated private cloud platform based on XCP-NG and Xen-Orchestra (XOA). It is also designed to offer cloud-like functionalities like automated provisioning of virtual machines, VM lifecycle, resource monitoring, and live migration — but without complicating the matter, flexibility, and simplicity, and cost efficiency.

In contrast to conventional enterprise cloud systems (e.g. OpenStack, VMware ESXi), the MiniCloud Platform adds a new layer of custom controller which brings out programmable APIs to coordinate with the virtualization infrastructure. This separates user interactions access to the hypervisor underneath it and enables the automation of administrators or applications and workflows programmatically.

The platform combines a number of modern technologies:

- FastAPI to develop a high-performance RESTful API layer.
- Celery and Redis asynchronous and distributed tasks.
- PostgreSQL as a persistent storage of metadata and metrics.
- XCP-NG and XOA VM operations including creation, metrics and migration

These elements build a united and scalable ecosystem that transforms a conventional virtualization configuration into a programmable mini-cloud infrastructure.

3.2 High-Level Functional Overview

The Mini-Cloud Platform is on a high level of operation in that it is an orchestration and automation layer on the XCP-NG hypervisor pool. The system is a modular architecture with every element being devoted to a particular functionality:

- **FastAPI Controller**: Acts as the entry point for all user or system requests. There are API endpoints for actions like creating a VM, deleting a VM, migrating one, and getting metrics.
- **Celery Workers (Async Task System)**: It processes long running or intensive actions (for example, cloning a VM template, migrating VMs) in a way that does not block API responses. Actions are performed in parallel across multiple workers.

- **Redis Message Broker**:This acts as a communication layer between the API and Celery workers. It queues and schedules tasks in a deterministic order.
- **PostgreSQL Database**: Stores metadata of VM, logs of API request, task state and performance metrics of XOA.
- **XCP-NG & XOA Integration Layer**: offers the real virtualization functions:
  - Clone from template
  - Attach to networks
  - Power related operations
  - Retrieval of metrics
  - Executing live migrations operations
- **Scheduler Tasks**: Run some background tasks periodically (for example, some metrics (like every minute), or cleanup tasks (like hourly))
- **Logging and Monitoring Framework**: Logs controller activity, worker execution logs, API requests, response and error traces.

When the user/service sends request (For example to create a VM), the Controller will validate the input, call a celery task, initiate the VM provisioning process via the XCP-NG/XOA connectors, and store status and metadata in the database. All this happens automatically.

3.3 User Roles

The Mini-Cloud Platform is principally an orchestration platform; however, it has implicit definitions of user roles and system actors:

3.3.1 Administrator

- Administers the XCP-NG servers underneath
- Manages the XOA instance
- Controls network, storage and host resources
- Utilizes the platform to create VMs for lab or production deployments

3.3.2 API Client

All services, automation scripts or user applications that communicate with the FastAPI controller are an API Client.

This role:

- Makes API calls to create, start, stop or migrate a VM.
- Get metrics or VM information.
- Consolidates platform with other tools (CI/CD, dashboards, etc.)

3.3.3 Controller Service

The rational player of the platform. It:

- Retrieves incoming requests
- Checks and validates parameters
- Drives work to Celery employees.
- Stores data in PostgreSQL database

3.3.4 Celery Worker Nodes

The functions of these nodes are:

- Performing long and heavy-weight processes.
- Re-attempting failed tasks
- Communicating the results of the tasks to the controller.

They can be scaled horizontally and that is, they can be executed on multiple machines by numerous worker processes.

3.3.5 XCP-NG Hypervisor Hosts

These servers run and perform the real virtualization methods:

- Running VMs
- Cloning templates
- Handling migrations
- Providing compute, network, and storage resources

3.3.6 XOA (Xen-Orchestra) Backend

Acts as an API gateway for:

- Retrieval of VM related metrics
- Live migration Orchestration.
- Media network and storage management.

3.4 Features Implemented

The Mini-Cloud Platform has a wide range of cloud-like features, which allows a seamless and automated workflow virtualization.

3.4.1 Virtual Machine Creation

- Automatic provisioning of VMs using pre-derived templates.
- Allows being specific with CPU, RAM, disk size and network.
- operates through XOA/XCP-NG API and xe CLI.
- Responsiveness is provided by asynchronous execution.
- VM Save VM information in PostgreSQL to use later.

3.4.2 Virtual Machine Management

Covers full lifecycle operations:

- Start
- Stop
- Suspend
- Resume
- Reboot
- Delete

These tasks are implemented via the controller with pre-installed logging as well as error handling.

3.4.3 Template Handling

- Displays the VM templates that are available in XCP-NG.
- Clones VMs from templates selected by the user.
- Validates and maps template UUIDs correctly.
- Aids in cloning standardized lab or development systems.

3.4.4 Metrics Collection

A dedicated module fetches VM metrics at regular intervals:

- CPU utilization
- Memory usage
- Disk I/O
- Network I/O

These metrics are:

- Data is collected through XOA's metrics API
- The data is stored in PostgreSQL
- The data can be presented for visualization or analytics

3.4.5 VM Migration

Facilitates live migration and storage-aware migration with XOA APIs:

- Automatically chooses or accepts target hosts
- Supports block migrations
- Achieves minimal downtime and data consistency
- Applicable for load balancing, tiresome maintenance, or general performance tuning

3.4.6 Centralized API System

The framework exposes:

- Clean and structured documented API endpoints
- JSON input and output envelopes
- Streaming speed in execution based on FastAPI
- Authentication hooks (optional for future extensions)

System APIs abstract the entire virtualization stack to become a programmable cloud interface.

3.4.7 Scheduler for Background Tasks

Includes scheduled jobs:

- For unattended periodic collection of VM metrics
- Automatic cleanup of ephemeral resources
- Daily or hourly health checks on the stack
- Background logging and reporting jobs

Jobs work in tandem to create the appearance of a micro-cloud management platform.

3.4.8 Monitoring and Logging

Comprehensive logging is executed throughout the framework:

- API interaction logs
- Celery worker execution logs
- Errors/exceptions logs
- VM provisioning and migration history logs

Log access creates transparency, debug-ability, and operational observability.

4. Project Design

4.1 Complete System Architecture

Mini-Cloud Platform design is based on the creation of a scalable, automated and modular cloud orchestration platform. The project is based on a multi-layered architecture which has separated the user interface, controller logic, asynchronous processing, virtualization management, compute operations, and storage access. The essence of the design philosophy is to make sure that each subsystem is able to scale automatically, is loosely coupled and is resilient even when it is heavily loaded.. The system architecture of the system is divided into two complementary diagram perspectives (1)A high-level architecture diagram of the system that models the end-to-end process flow of VM provisioning, scheduling, and hypervisor operations; and (2) A layer-wise architecture diagram of the system that sub-divides its system system into functional layers including User Layer, Controller Layer, Orchestration Layer, Compute Layer, and Storage Layer.

4.1.1 Architectural Overview Diagram

The following diagrams represent both perspectives of the system:

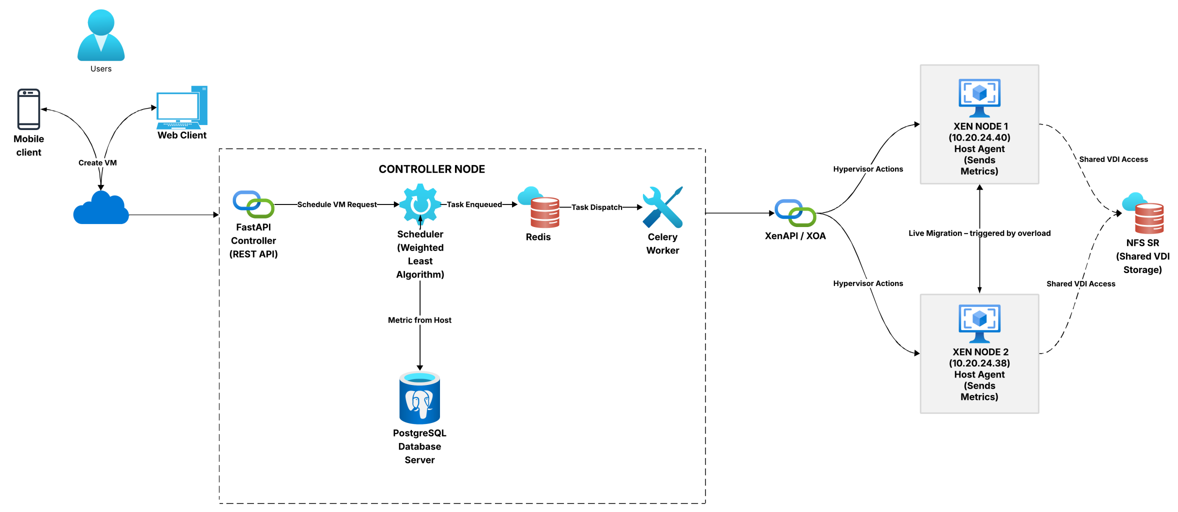

Users
Mobile client
Web Client
Create VM
CONTROLLER NODE
FastAPI Controller (REST API)
Schedule VM Request
Scheduler (Weighted Least Algorithm)
Task Enqueued
Redis
Task Dispatch
Celery Worker
Metric from Host
PostgreSQL Database Server
XenAPI / XOA
Hypervisor Actions
Hypervisor Actions
XEN NODE 1 (10.20.24.40) Host Agent (Sends Metrics)
Live Migration – triggered by overload
XEN NODE 2 (10.20.24.38) Host Agent (Sends Metrics)
Shared VDI Access
Shared VDI Access
NFS SR (Shared VDI Storage)

Figure 4.1: High-Level Architecture Diagram of the Mini-Cloud Platform

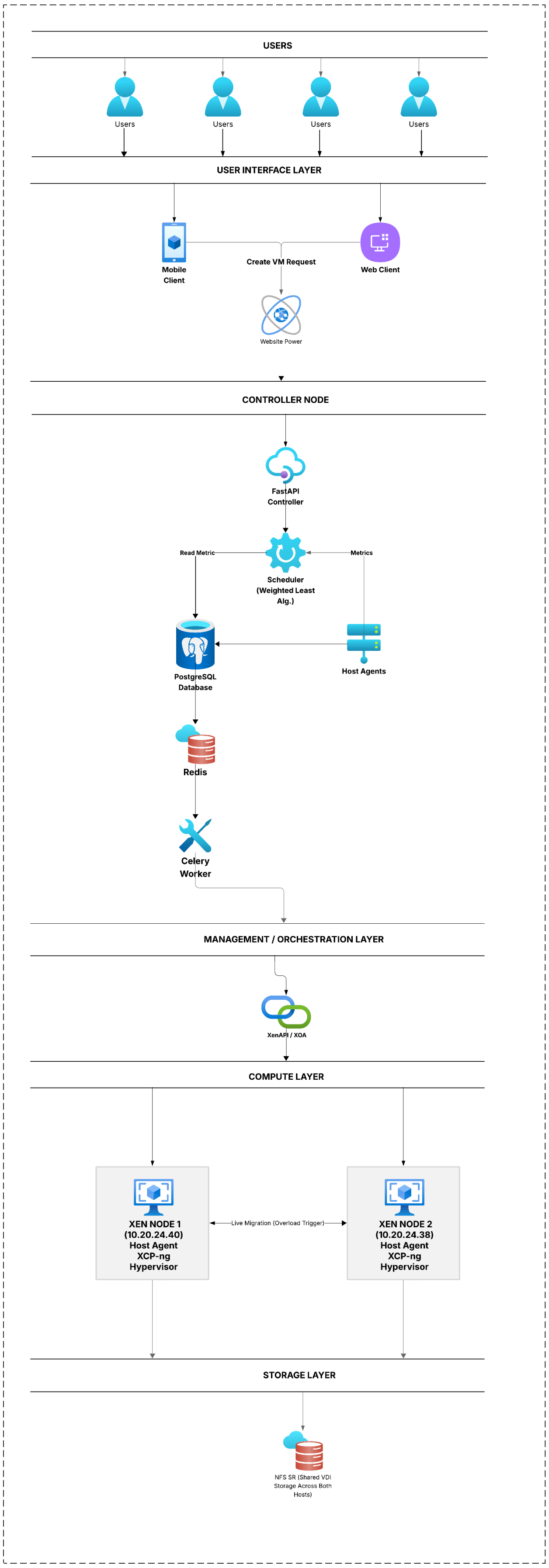

USERS
Users
Users
Users
Users
USER INTERFACE LAYER
Mobile Client
Create VM Request
Web Client
Website Power
CONTROLLER NODE
FastAPI Controller
Read Metric
Metrics
Scheduler (Weighted Least Alg.)
PostgreSQL Database
Host Agents
Redis
Celery Worker
MANAGEMENT / ORCHESTRATION LAYER
XenAPI / XOA
COMPUTE LAYER
XEN NODE 1 (10.20.24.40) Host Agent XCP-ng Hypervisor
Live Migration (Overload Trigger)
XEN NODE 2 (10.20.24.38) Host Agent XCP-ng Hypervisor
STORAGE LAYER
NFS SR (Shared VDI Storage Across Both Hosts)

Figure 4.2: Layer-wise Architecture of the Mini-Cloud Platform

Your system architecture consists of six distinct layers, each with a specific purpose. Below is the detailed explanation that corresponds to the layer-wise diagram:

(A) Users Layer

This layer consists of the end-users who request deployment or management of virtual machines. These users may be students, researchers, system administrators, or automation scripts. Users do not interact with the hypervisors directly; all interactions go through the API.

(B) User Interface Layer

This layer handles request submission to the cloud platform. It may include a mobile client, web client, or a future dashboard. Users submit actions such as creating VMs, deleting VMs, viewing VM status, or initiating migrations. The UI sends HTTP API requests and contains no business logic.

(C) Controller Node Layer

This is the brain of the Mini-Cloud platform. It contains:

- **FastAPI Controller (REST API)** – Primary point point of all operations. It authenticates API requests, captures metadata, and sends Celery jobs
- **Scheduler (Weighted Least Algorithm)** – Chooses the least loaded hypervisor based on PostgreSQL host metrics.
- **PostgreSQL Database** – Metadata of VM, metrics, logs, task entries, and migration history.
- **Redis** – Celery task queuing message broker.
- **Celery Worker** – This performs heavy tasks like VM creation, deletion, metricals, and migration.

(D) Management / Orchestration Layer

the layer that handles management and coordination of various system layers. The coordination and management of different layers of the system is also dealt with in the management / Orchestration Layer.Its main constituent is the XenAPI/XOA, which runs the operations of VM lifecycle, clones templates, retrieves metrics and both live VM migration..

(E) Compute Layer (Hypervisor Layer)

XCP-ng hypervisors can be found in this layer:

- XEN Node 1 (10.20.24.40)
- XEN Node 2 (10.20.24.38)

Both nodes have virtual machines and offer compute and networking capabilities and report the performance to the controller.

(F) Storage Layer

In this layer, there is the NFS Storage Repository (Shared VDIStorage) that enables the shared block storage between all hypervisors. Shared storage facilitates live migration that eliminates the need to copy disk information between hosts.

4.1.2 Component Responsibilities

Below is the detailed breakdown of responsibilities for each component:

FastAPI Controller

- Checks and authenticates the incoming API requests.
- The external interface of the cloud.
- Sends long term operations to Celery workers.
- Writes and reads VM metadata in PostgreSQL.

Scheduler (Weighted Least Algorithm)

- Reads contain postgreSQL metrics.
- Calculates the load based on weighted variables: CPU, RAM, network, running VMs.
- Precludes hypervisor congestion through the ideal host selection.

Redis Message Broker

- Queues asynchronous tasks.
- Makes implementation not blocking.
- Favors job distribution that is scalable.

Celery Worker

- Performs processor-intensive and lengthy tasks.
- Provides VM provisioning, migrates, deletion and metrics.
- Conducts error management and logs.

XenAPI / XOA

- Carries out VM operations like start and lead, re-booth and clone.
- Administrates real-time migration of people.
- Directs metrics and information about hypervisors to the controller.

Hypervisor Nodes

- Hypervisor- virtual Governor Run virtual machines and assign resources (CPU, RAM, and network).
- Send controller report measures through the host agents.
- Perform live migrations.

PostgreSQL Database

- Storage Service VM metadata, migration records, history of metrics, template data, etc.
- Ensures analytics and dashboards.

NFS Storage Repository

- Offers common VDI storage that can be used by a number of hypervisors.
- Allows real-time migration of VMs.

4.1.3 Data Flow Explanation

(A) VM Creation Request Flow

1. User sends Create VM request on UI to FastAPI.
2. The FastAPI verifies the request and sends it forward to the Scheduler.
3. Scheduler asks PostgreSQL to give it host metrics.
4. Scheduler uses weighted least algorithm to choose a host of optimal choice.
5. Scheduler adds job to Redis queue.
6. Celery Worker removes the tasks and interacts with XenAPI/XOA.

7. XOA does VM provisioning in the chosen hypervisor.
8. Commit Metadata PostgreSQL is updated by the worker.
9. Success response is sent to the user by API.

(B) Metrics Collection Flow

1. Open-ended metrics are issued to Scheduler by Hypervisor Host Agents.
2. Scheduler keeps the measures in PostgreSQL.
3. Celery Workers get the metrics of XOA at predetermined timelines.
4. Analytics are logged in metrics.

(C) VM Migration Flow

1. Scheduler uses metrics to detect overloaded host.
2. Celery migration task is triggered by a migration API request.
3. Celery Worker orders the migration of VM to a different host.
4. In hypervisor, migration is performed live with the help of shared storage.
5. The migration data is logged into PostgreSQL.

(D) Storage Access Flow
The Four hypervisors share the same NFS Shared Repository which supports:

- Shared access to VDI files
- Live migrations with near to zero-downtime.
- Stability: VM failover/movement.

4.2 Logical Design

The logical design phase explains the inside design of the Mini-Cloud Platform in the form of classes, relationships, data models, and major structural elements of the architecture. It is the way the system is internally arranged - regardless of its physical implementation. Logical design gives the assurance that every single component has its assigned responsibilities, well-defined input and output, and the way these other components can interact.

The subsection deals with an object-oriented abstraction of the platform through Class Diagram. The class diagram reflects the key structures that are engaged in API processing, asynchronous task execution, VM settlement, metrics acquisition, planning, and database communications. Every class contains a collection of duties, qualities and roles that jointly assigns the aggregate capability of the cloud orchestration framework.

4.2.1 Class Diagrams

The Class Diagram of the Mini-Cloud Platform is further subdivided into four conceptual clusters including API Layer Classes, Scheduler Layer Classes, Worker Layer Classes and Database Layer Classes. The combination of these classes determines the gist of the logic, and interactions amongst the controller and the scheduler, workers and the underlying infrastructure.

API Layer Classes

FastAPIController It is a class that processes incoming HTTP requests and has an incoming request body which is validated and interacts with the Scheduler and allocates work to the Celery Worker, and makes database operations.
Responsibilities:

- Accept as well as authorize API requests.
- Map requests into internal operations.
- Dispatch Celery tasks
- Update and query PostgreSQL database

Key Methods:

- `create_vm()`
- `delete_vm()`
- `migrate_vm()`
- `get_vm_status()`
- `get_metrics(vm_id)`

RequestSchema (Pydantic Models) Defines the structure of incoming JSON requests for VM creation, deletion, migration, or metrics queries.
Attributes:

- VM name
- CPU count
- RAM size
- Disk size
- Template ID
- Host (optional)

Scheduler Layer Classes

Scheduler This class implements the Weighted Least Algorithm, responsible for selecting the most optimal hypervisor for VM creation.
Attributes:

- host_metrics
- weight_cpu
- weight_ram
- weight_network
- weight_vm_count

Methods:

- `select_best_host()`
- `fetch_latest_metrics()`
- `calculate_weighted_score()`

HostMetrics Represents the metrics collected from hypervisors. These values are used by the Scheduler to make host selection decisions.
Attributes:

- cpu_usage
- ram_usage
- network_io
- running_vms

Worker Layer Classes

CeleryWorker Executes background tasks using Celery Workers. These tasks include VM provisioning, migration, metrics collection, and deletion. **Methods include:**

- `task_create_vm()`
- `task_migrate_vm()`
- `task_collect_metrics()`
- `task_delete_vm()`

XenAPIClient Responsible for communication with XCP-ng/XOA for performing hypervisor actions.
Methods:

- `clone_template()`
- `configure_vm_resources()`
- `attach_network()`
- `power_on_vm()`
- `migrate_vm()`
- `fetch_metrics()`

Database Layer Classes (SQLAlchemy Models)

VMModel Stores metadata related to each virtual machine.
Attributes:

- vm_uuid
- template_id
- cpu
- ram
- disk
- host
- status

MetricsModel Stores performance metrics collected periodically.
Attributes:

- cpu_usage
- ram_usage
- disk_io
- network_io
- timestamp

TaskLogModel Represents execution logs for Celery tasks.
Attributes:

- task_id
- vm_id
- task_type
- status
- error_message
- timestamp

Class Diagram Placeholder

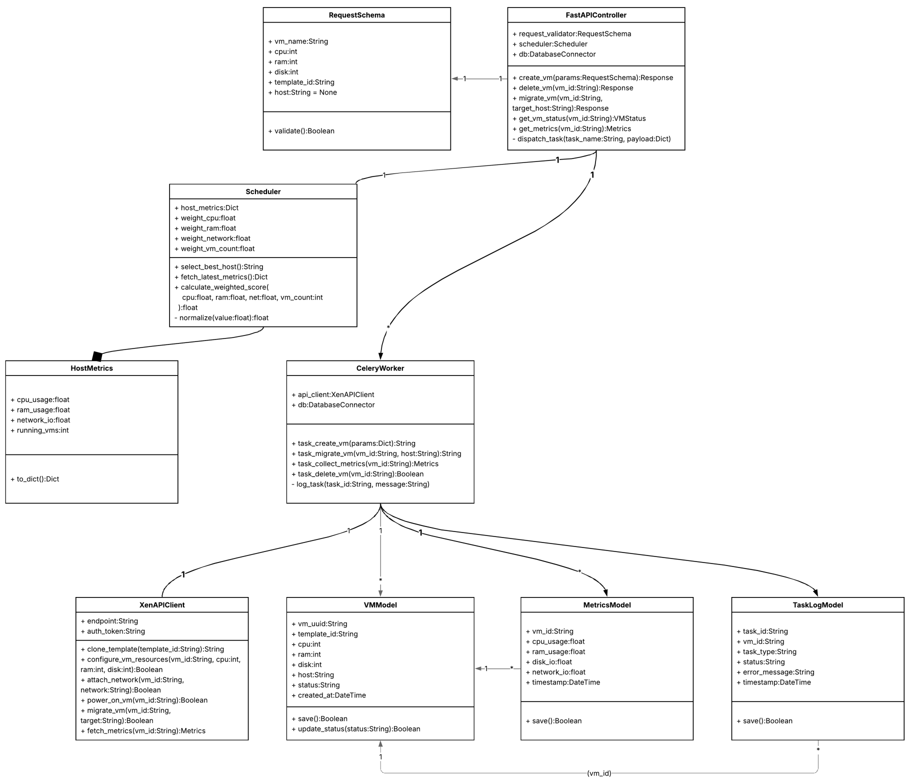

RequestSchema
+ vm_name:String
+ cpu:int
+ ram:int
+ disk:int
+ template_id:String
+ host:String = None
+ validate():Boolean
FastAPIController
+ request_validator:RequestSchema
+ scheduler:Scheduler
+ db:DatabaseConnector
+ create_vm(params:RequestSchema):Response
+ delete_vm(vm_id:String):Response
+ migrate_vm(vm_id:String, target_host:String):Response
+ get_vm_status(vm_id:String):VMStatus
+ get_metrics(vm_id:String):Metrics
- dispatch_task(task_name:String, payload:Dict)
Scheduler
+ host_metrics:Dict
+ weight_cpu:float
+ weight_ram:float
+ weight_network:float
+ weight_vm_count:float
+ select_best_host():String
+ fetch_latest_metrics():Dict
+ calculate_weighted_score(cpu:float, ram:float, net:float, vm_count:int):float
- normalize(value:float):float
HostMetrics
+ cpu_usage:float
+ ram_usage:float
+ network_io:float
+ running_vms:int
+ to_dict():Dict
CeleryWorker
+ api_client:XenAPIClient
+ db:DatabaseConnector
+ task_create_vm(params:Dict):String
+ task_migrate_vm(vm_id:String, host:String):String
+ task_collect_metrics(vm_id:String):Metrics
+ task_delete_vm(vm_id:String):Boolean
- log_task(task_id:String, message:String)
XenAPIClient
+ endpoint:String
+ auth_token:String
+ clone_template(template_id:String):String
+ configure_vm_resources(vm_id:String, cpu:int, ram:int, disk:int):Boolean
+ attach_network(vm_id:String, network:String):Boolean
+ power_on_vm(vm_id:String):Boolean
+ migrate_vm(vm_id:String, target:String):Boolean
+ fetch_metrics(vm_id:String):Metrics
VMModel
+ vm_uuid:String
+ template_id:String
+ cpu:int
+ ram:int
+ disk:int
+ host:String
+ status:String
+ created_at:DateTime
+ save():Boolean
+ update_status(status:String):Boolean
MetricsModel
+ vm_id:String
+ cpu_usage:float
+ ram_usage:float
+ disk_io:float
+ network_io:float
+ timestamp:DateTime
+ save():Boolean
TaskLogModel
+ task_id:String
+ vm_id:String
+ task_type:String
+ status:String
+ error_message:String
+ timestamp:DateTime
+ save():Boolean
(vm_id)

Figure 4.3: Class Diagram of the Mini-Cloud Platform

4.2.2 Entity–Relationship (ER) Diagram for Database

Entity Relationship (ER) model depicts the logical design of the database of the Mini-Cloud Platform. It depicts the relationship between the basis data entities, which are the virtual machines, performance measurements, task records and VM templates. This ER architecture allows the storage, access, and management of metadata created during the VM lifecycle, background tasks processing, metrics gathering and migration activities.

The ER diagram documents the essence of the interrelationship between the entities in the form of primary and foreign keys and cardinality constraints which would guarantee referential integrity throughout the system.

Entities and Attributes

1. VMModel (Virtual Machine Entity) Refers to a VM that is being created in the platform.

- **vm_uuid (PK)** – Unique identifier for the VM.
- template_id – Reference to the VM template used.
- cpu – Allocated vCPUs.
- ram – Allocated RAM in MB.
- disk – Disk size in GB.
- host – Host hypervisor on which the VM is deployed.
- status – Current VM state (running, stopped, suspended, migrating).
- created_at – Timestamp of VM creation.

2. MetricsModel (Performance Metrics Entity) Stores time-series metrics for each VM.

- **id (PK)**
- **vm_id (FK)** – References VMModel.vm_uuid
- cpu_usage – CPU utilization percentage.
- ram_usage – RAM usage in MB.
- disk_io – Disk I/O throughput.
- network_io – Network utilization.
- timestamp – Time at which the metric was recorded.

3. TaskLogModel (Task Execution Logs Entity) Represents logs for Celery worker task executions.

- **id (PK)**
- **vm_id (FK)** – References VMModel.vm_uuid
- task_type – Type of task executed (create, delete, migrate, metrics).
- status – Task status (success, failure, running).
- error_message – Message associated with failures.
- timestamp – Execution timestamp.

4. TemplateModel (Optional) Represents VM templates stored or registered within the environment.

- **template_id (PK)**
- template_name
- os_type

Relationships and Cardinalities

- **VMModel (1) — (M) MetricsModel** Each VM can have many performance metric entries collected over time. The foreign key `vm_id` in MetricsModel references `vm_uuid` in VMModel.
- **VMModel (1) — (M) TaskLogModel** Each VM generates many task logs such as creation, migration, deletion, or metrics collection tasks. The foreign key `vm_id` in TaskLogModel references `vm_uuid` in VMModel.
- **TemplateModel (1) — (M) VMModel** (if template tracking is enabled) Multiple VMs can be created using the same template. The attribute `template_id` in VMModel references TemplateModel.template_id.

Together, these entities form the core of the database system supporting the Mini-Cloud orchestration platform. The design ensures high consistency, efficient querying, and clear traceability between VM lifecycle operations and their associated metrics and logs.

ER Diagram Placeholder

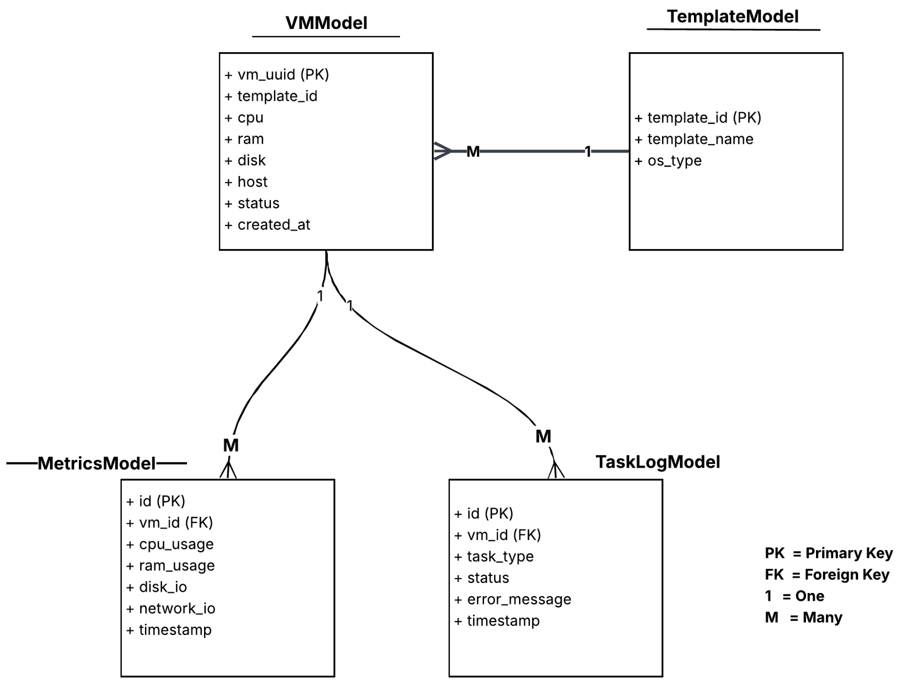


Figure 4.4: Entity–Relationship Diagram for the Mini-Cloud Platform

4.2.3 API Layer Logical Design

The API Layer is the side that leads to the Mini-Cloud Platform. It isolates the complexity of the direct interaction relative to the hypervisor and offers a programmable, clean and secure interface to the creation, management, and monitoring of the virtual machines. The layer is implemented on the FastAPI design pattern that adheres to the REST design pattern that guarantees high performance, validation of requests, concurrency, and scalability.

The API Layer is mainly used to map the external user facing HTTP request towards the internal operations that are managed by the scheduler, worker system, and database systems.

Roles of the API Layer

The responsibility of API layer is to:

1. Request Handling

- The ability to receive structured JSON requests by the web/mobile clients.
- Supervising inbound payload by Pydantic models.
- Securing type safety and schema verification.

2. Routing & Endpoint Management Each cloud action maps to a specific API endpoint:

- `/vm/create`
- `/vm/delete/{vm_id}`
- `/vm/migrate/{vm_id}`
- `/vm/status/{vm_id}`
- `/metrics/{vm_id}`

3. Interaction with Scheduler The API Layer communicates with the Scheduler to:

- Fetch fresh host metrics.
- Select the optimal hypervisor.
- Trigger decisions for VM placement.

4. Asynchronous Task Dispatching The API Layer does not execute heavy operations. Instead, it creates Celery tasks for:

- VM provisioning
- VM deletion
- Migration
- Metrics collection

This is done via a task dispatcher that pushes requests to Redis.

5. Database Integration The API interacts with PostgreSQL to:

- Insert new VM records

- Update VM states
- Query VM status
- Retrieve metrics data
- Log API activity

6. Response Management The API returns:

- Task IDs for long-running operations
- Structured JSON responses
- Error handling messages
- Execution results

All responses follow consistent status codes and schema definitions.

Core Components in the API Layer

1. FastAPI Controller The controller exposes endpoints and orchestrates all incoming API calls.

2. RequestSchema (Pydantic Models) Ensures:

- Automatic validation
- Type enforcement
- Error reporting
- Clean documentation via OpenAPI

3. API Router Organizes endpoints by logical grouping:

- VM operations
- Metrics endpoints
- Migration endpoints

4. Task Dispatcher Handles the transformation of API calls into asynchronous Celery tasks.

Logical Flow of an API Request

1. Client → API Gateway User sends a request (e.g., create VM).

2. Pydantic Validation Payload schema is validated. If invalid → error returned immediately.

3. Scheduler Invocation API requests:

- Host list
- Metrics
- Best-host selection

4. Celery Task Creation A task is queued in Redis containing:

- VM configuration
- Selected host
- Template UUID
- Metadata

5. Database Logging `VMModel` or `TaskLogModel` entries updated.

6. Response API returns:

- Success message
- Task ID
- VM UUID (when ready)

Advantages of This Logical Design

1. Scalability The API is stateless and horizontally scalable.

2. Extensibility New endpoints can be added without affecting worker logic.

3. Fault Tolerance Worker failures do not affect API responsiveness.

4. Clear Separation of Concerns API only handles:

- Validation
- Routing
- Dispatch

Execution occurs in Celery workers.

5. **Security** Pydantic + FastAPI ensure:

- Input sanitization
- Automatic error filtering
- Reduced attack vectors

API Layer Logical Flow Diagrams

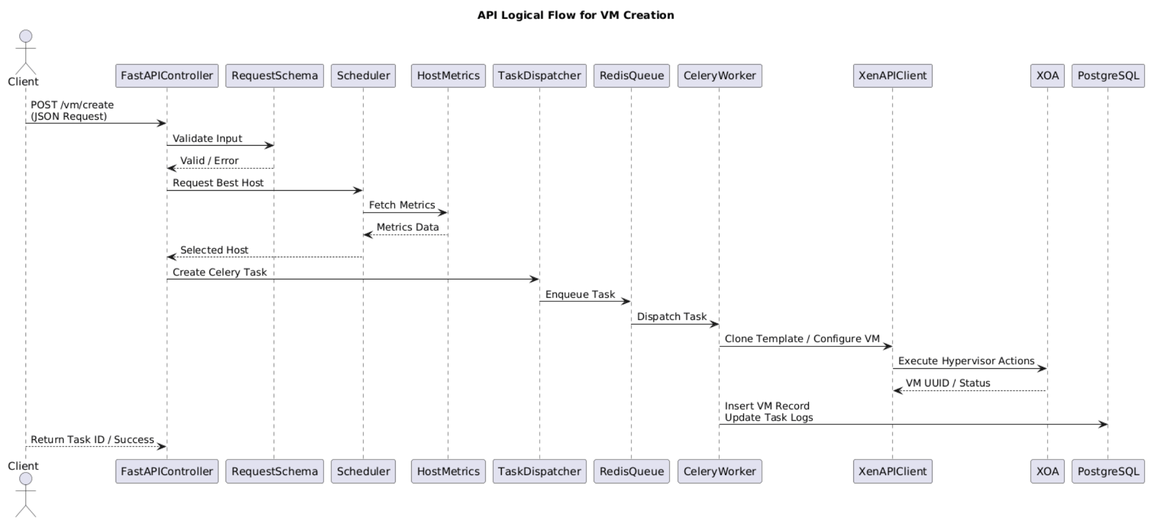


Figure 4.5: API Layer Logical Flow: Sequence Diagram Representing Request-to-Execution Flow

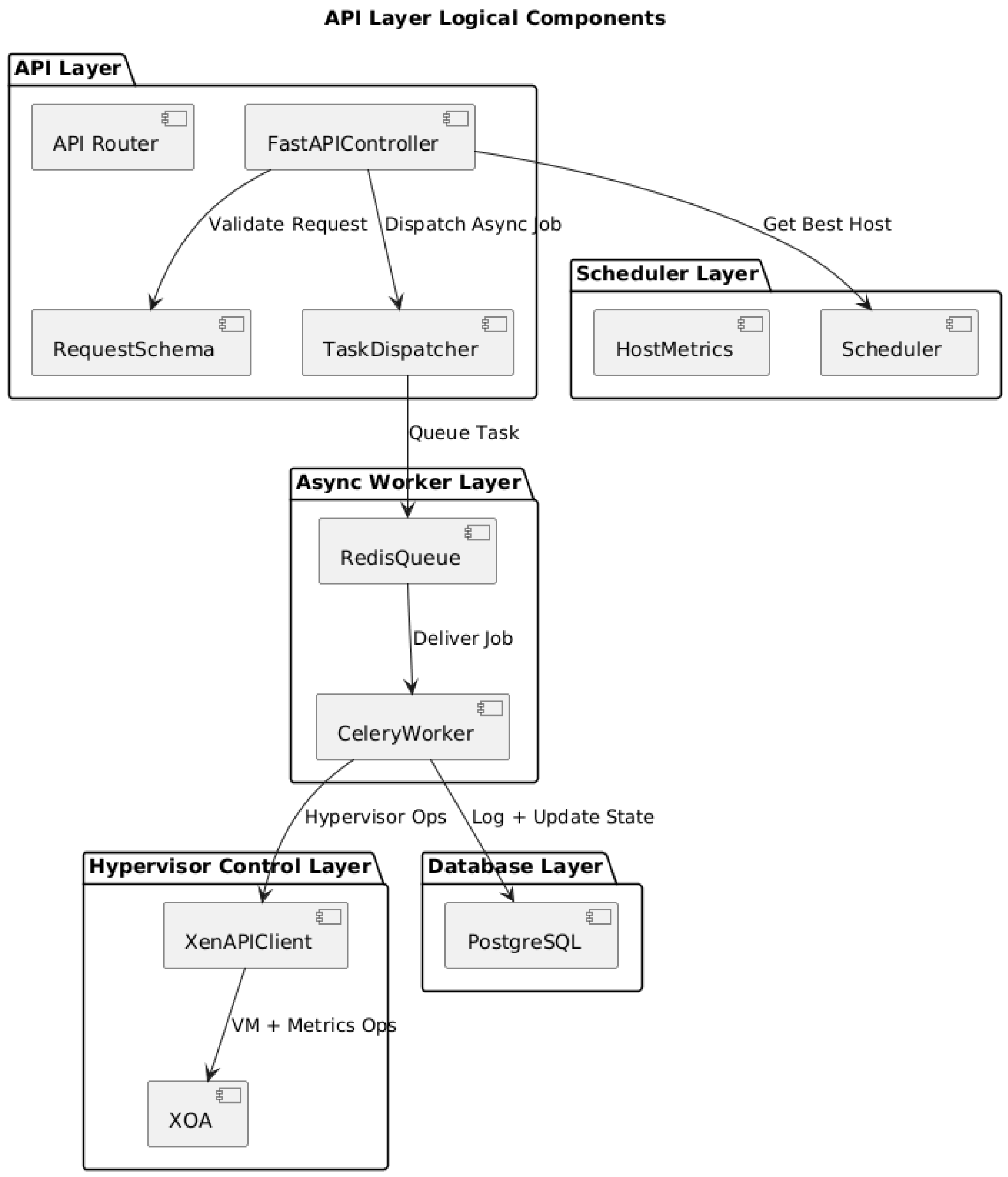


Figure 4.6: API Layer Logical Component Diagram Showing Internal System Interactions

4.2.4 Microservices Interaction Design

The Mini-Cloud Platform follows a microservices-inspired architecture in which every major subsystem operates independently and communicates through well-defined interfaces. This design ensures scalability, fault tolerance, modularity, and clear separation of concerns across the platform. The interaction model between these microservices governs the entire lifecycle of virtual machines, including provisioning, migration, task execution, monitoring, and performance metrics collection.

Each microservice is responsible for a specific set of operations while interacting with others using lightweight communication mechanisms. This allows the platform to scale individual components without affecting the system as a whole and ensures that failures in one module do not propagate across the system.

Core Microservices and Their Roles

1. API Service (FastAPI Controller) Acts as the entry point to the system. It validates incoming requests, interacts with the scheduler, creates asynchronous tasks, updates the database, and returns responses to users. It does not perform any long-running operations but delegates such tasks to the Celery Workers.

2. Scheduler Service Implements the Weighted Least Algorithm to determine optimal host selection for VM placement. It accesses host performance metrics from the database and returns the best hypervisor node based on weighted scoring.

3. Task Dispatch Service (Redis Queue) Redis is used as a lightweight message broker through which asynchronous tasks are queued. It enables decoupled communication between the API service and Celery Workers, allowing the system to scale horizontally.

4. Celery Worker Service Executes all heavy and long-running operations, including VM provisioning, migration, VM deletion, and metrics retrieval. Workers communicate with the XenAPI/XOA service to perform actual hypervisor-level actions.

5. XenAPI/XOA Service This external service performs virtualization-layer actions such as template cloning, VM instantiation, migration, network attachment, and metrics retrieval. It exposes an API that the Celery Worker invokes to interact with the XCP-ng hypervisor pool.

6. Database Service (PostgreSQL) Stores VM metadata, template information, task logs, and performance metrics. All microservices interact with the database either for read or write operations.

Microservice Interaction Workflow

1. Client Request A user initiates an operation (e.g., VM creation) through the API. The request is validated and forwarded to the scheduler.

2. Host Selection The scheduler computes weighted scores for each host based on CPU usage, RAM usage, network utilization, and VM count. The optimal host is returned to the API.

3. Task Dispatch The API service creates a Celery task containing the VM configuration and scheduling decision. This task is queued into Redis.

4. Asynchronous Execution A Celery Worker pulls the job from Redis and communicates with the XenAPI/XOA service to clone templates, configure resources, start the VM, or perform migrations.

5. Database Update After task execution, the worker service logs results in PostgreSQL, storing new VM entries, metrics, or task history records.

6. Client Response The API returns a task identifier and success response to the user, with the option to check the VM state later.

Advantages of the Microservices Interaction Model

- **Loose Coupling:** Each service runs independently, allowing modular updates and isolated failures.
- **Scalability:** Celery Workers and API nodes can scale horizontally to meet demand.
- **Reliability:** Redis ensures durable message passing, enabling reliable asynchronous execution.
- **Fault Isolation:** Failures in one service do not disrupt the entire platform.
- **Extensibility:** New microservices (e.g., monitoring dashboards or autoscaling engines) can be added easily.

Microservices Interaction Diagram Placeholder

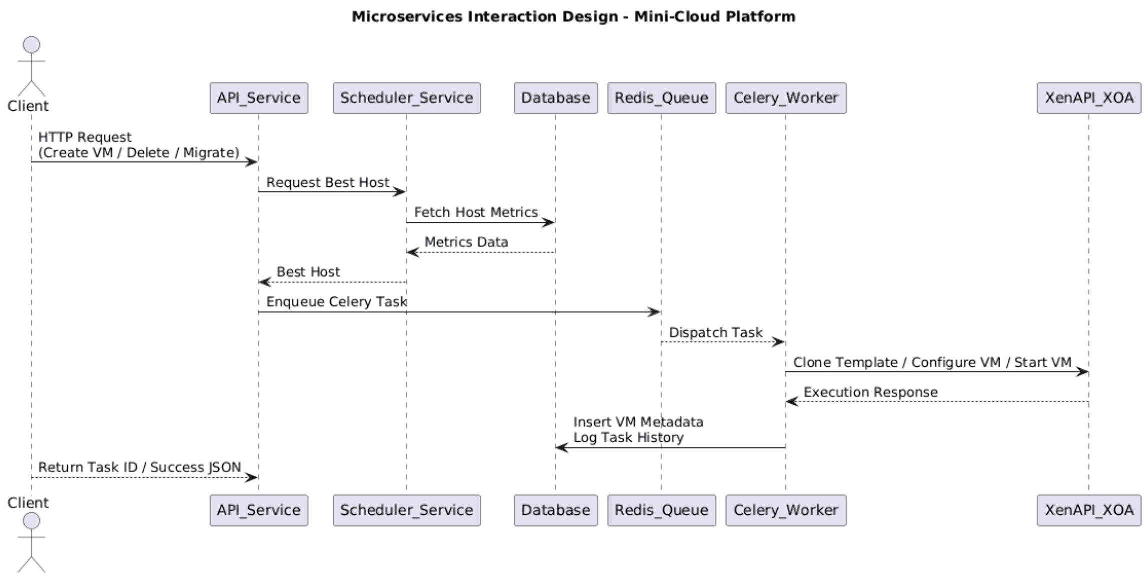


Figure 4.7: Microservices Interaction Diagram for the Mini-Cloud Platform

4.3 Behavioral Design

Behavioral design focuses on how different components of the Mini-Cloud Platform behave over time, how they interact with each other dynamically, and how the system responds to various events such as VM provisioning, migration, metrics collection, and scheduled background tasks. Unlike structural design, which describes the system's static architecture, behavioral design illustrates the runtime flow of control through the system using sequence diagrams and activity workflows.

This section presents detailed sequence diagrams that describe the interaction between the API service, scheduler, Celery workers, XenAPI/XOA, Redis, and the database during key operations of the Mini-Cloud Platform.

4.3.1 Sequence Diagrams

Sequence diagrams represent the chronological flow of operations in the system. They describe how objects interact with each other, the messages exchanged, the order of events, and the overall runtime behavior of the components involved.

This subsection contains sequence diagrams for:

- VM Creation Sequence
- VM Migration Sequence
- Metrics Collection Sequence
- Task Scheduling Sequence

VM Creation Sequence

The VM creation sequence describes the interactions involved in provisioning a new virtual machine. The workflow includes request validation, host selection, task dispatching, VM instantiation, and database updates.

- The client sends a VM creation request to the FastAPI Service.
- The API validates input using the RequestSchema.
- The API communicates with the Scheduler to determine the optimal hypervisor.
- A Celery task is created and queued into the Redis broker.
- A Celery Worker pulls the job and interacts with XenAPI/XOA to clone a VM template, configure its resources, and power on the VM.

- The worker updates the PostgreSQL database with VM metadata.
- A task ID is returned to the user for tracking progress.

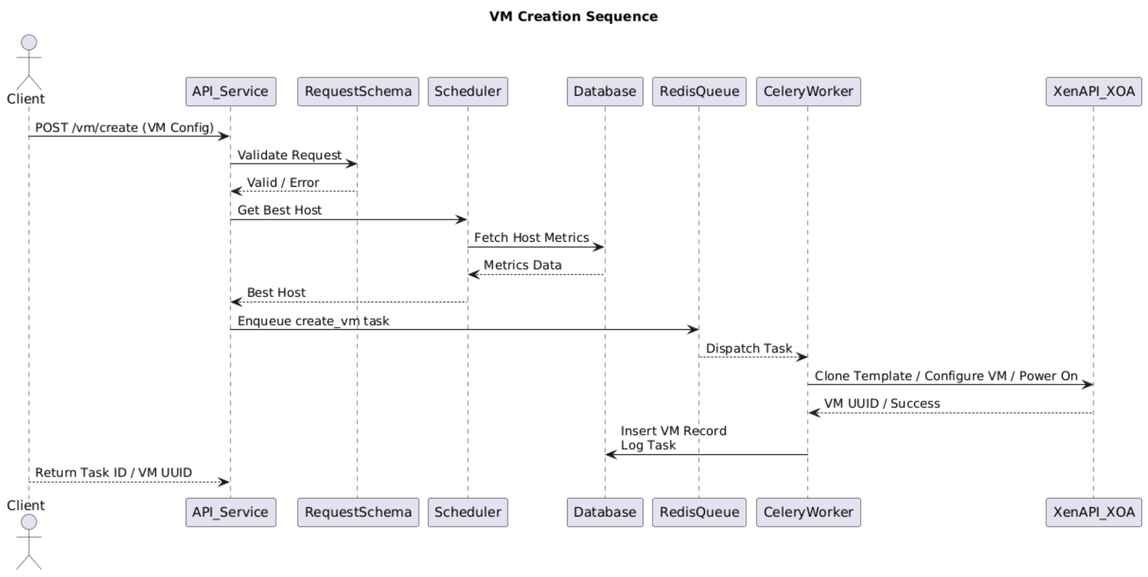


Figure 4.8: VM Creation Sequence Diagram

VM Migration Sequence

The VM migration sequence describes how the system handles live migration of a virtual machine from one host to another. This is essential for load balancing, maintenance, and optimized resource utilization.

- The user requests a VM migration.
- The API invokes the Scheduler to validate host suitability.
- A migration task is created and sent to Redis.
- A Celery Worker consumes the task and calls XenAPI/XOA to execute the live migration.
- Once completed, the worker updates the VM's host entry in PostgreSQL.
- The user receives a confirmation response.

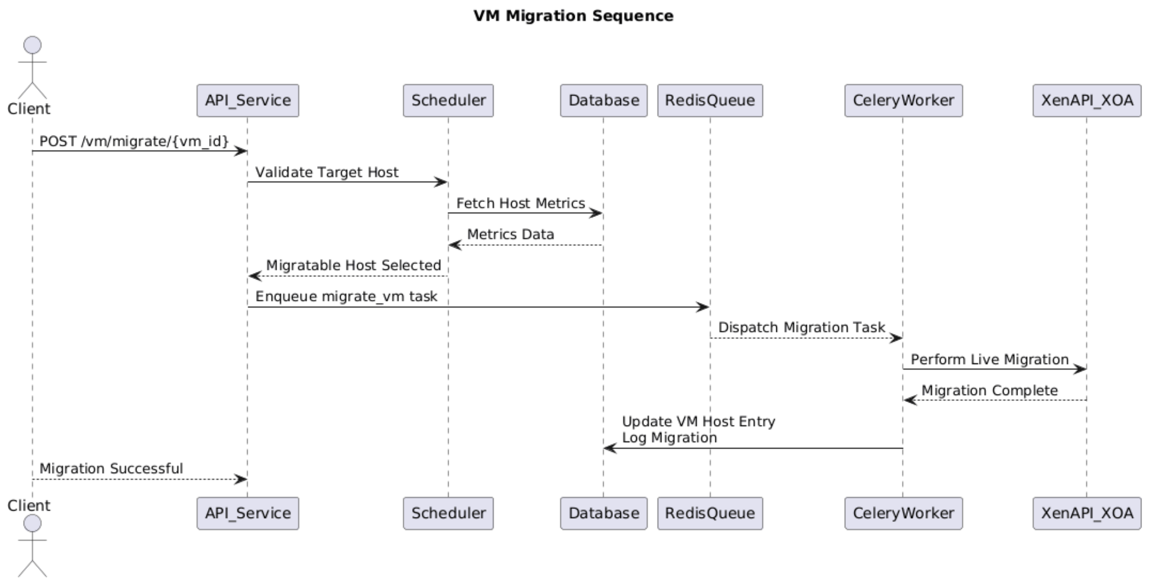


Figure 4.9: VM Migration Sequence Diagram

Metrics Collection Sequence

The metrics collection sequence describes how the system retrieves and stores VM performance metrics such as CPU usage, memory consumption, disk I/O, and network activity.

- The scheduler or Celery beat triggers a metrics collection task.
- The API or worker triggers a metrics fetch job.
- The task is queued into Redis.
- A Celery Worker retrieves metrics via XenAPI/XOA.
- The metrics are stored in PostgreSQL.

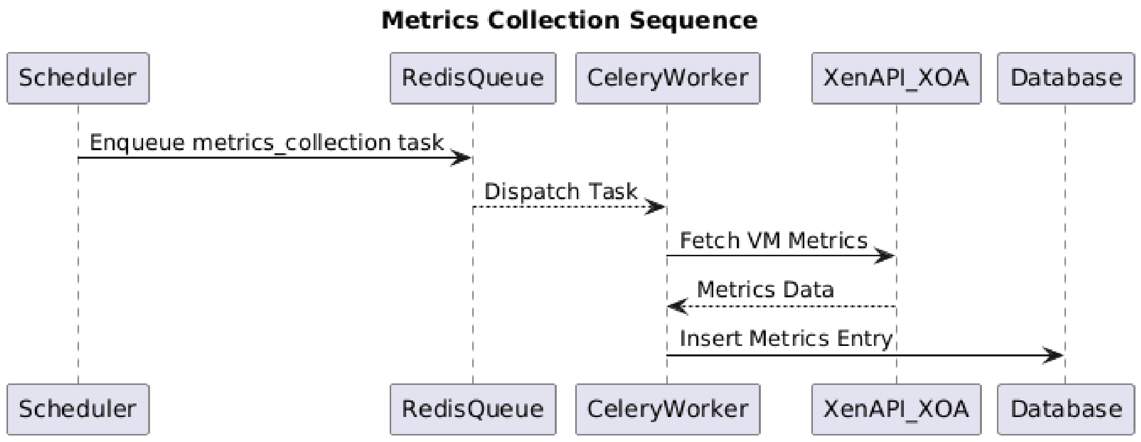


Figure 4.10: Metrics Collection Sequence Diagram

Task Scheduling Sequence

This sequence describes how periodic background tasks (e.g., metrics collection, cleanup activities) are scheduled and executed.

- Celery Beat triggers periodic tasks based on schedule configuration.
- Beat pushes the scheduled tasks into the Redis Queue.
- Celery Workers pick tasks and execute them asynchronously.
- Relevant updates are written to PostgreSQL.

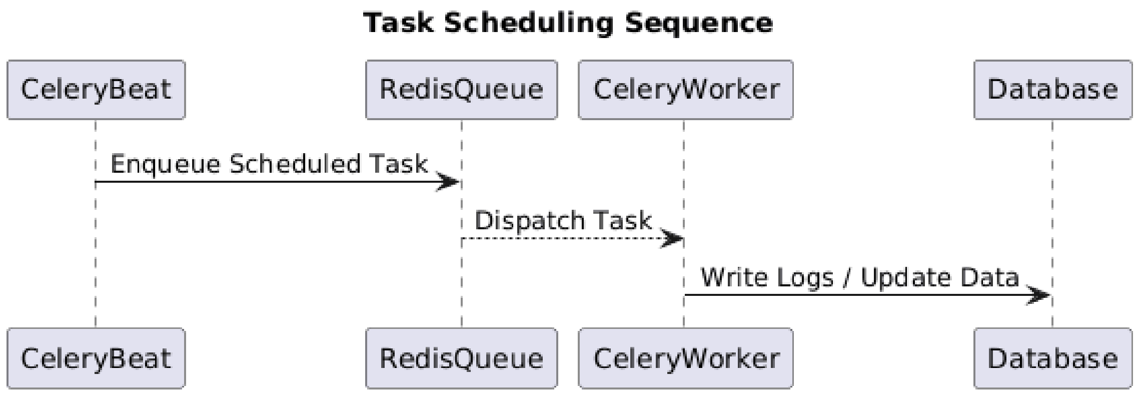


Figure 4.11: Task Scheduling Sequence Diagram

4.3.2 Activity Diagrams

Behavioral activity diagrams describe the major operational workflows inside the Mini-Cloud Platform. They capture the control flow, decision points, parallel activities, and termination conditions for common operations such as VM requests, monitoring/metrics collection, and asynchronous task execution. Activity diagrams complement sequence diagrams by showing branching, concurrency and high-level procedural logic.

VM Request Flow

The VM Request Flow activity diagram models the user-initiated VM creation process from request reception through validation, scheduling, task dispatching, provisioning, and final logging. Key activities include:

- Receive request (Client → FastAPI)
- Validate payload (Pydantic schema)

- Authenticate/Authorize (if enabled)
- Query Scheduler for host selection
- Enqueue task in Redis (Task Dispatcher)
- Execute provisioning in Celery Worker (clone template, configure, power on)
- Update database with VM metadata and task log
- Return response / notify client

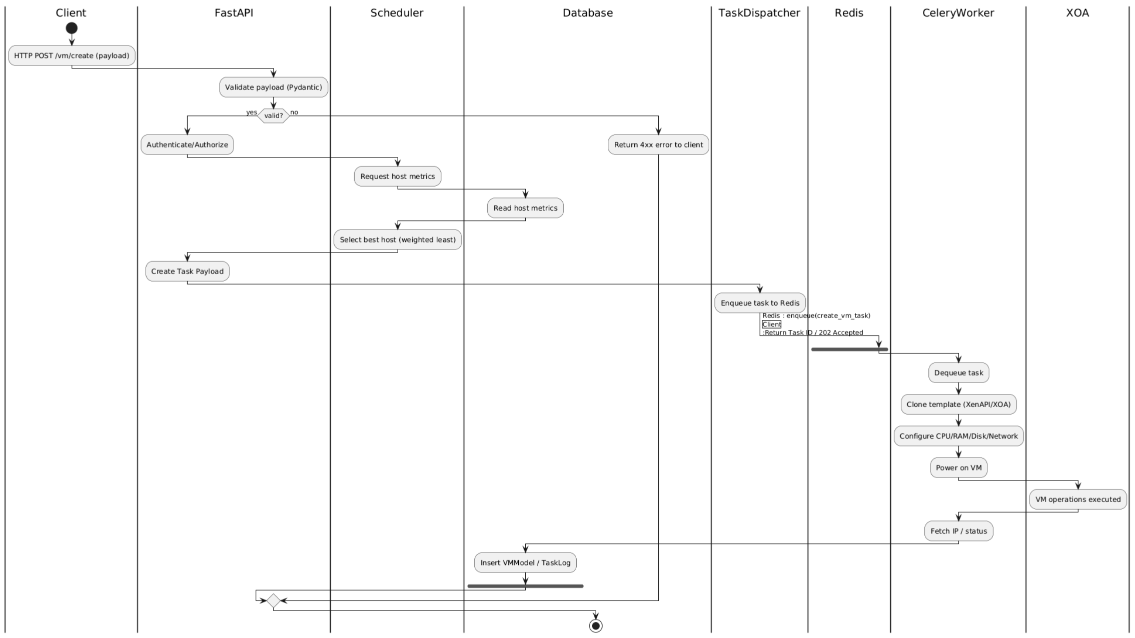


Figure 4.12: Activity Diagram: VM Request Flow

Monitoring / Metrics Flow

The Monitoring/Metrics Flow activity diagram shows periodic metrics collection and storage workflow. It includes scheduled trigger (Celery beat), parallel metric fetch for multiple VMs, aggregation and storage, and optional alerting when thresholds are exceeded. Important steps:

- Trigger scheduler (Celery beat or external scheduler)
- Build list of active VMs
- Parallel dispatch of metrics fetch tasks to Celery workers
- Collect metrics via XOA/XenAPI
- Store metrics in PostgreSQL

- Check thresholds / generate alerts (optional)

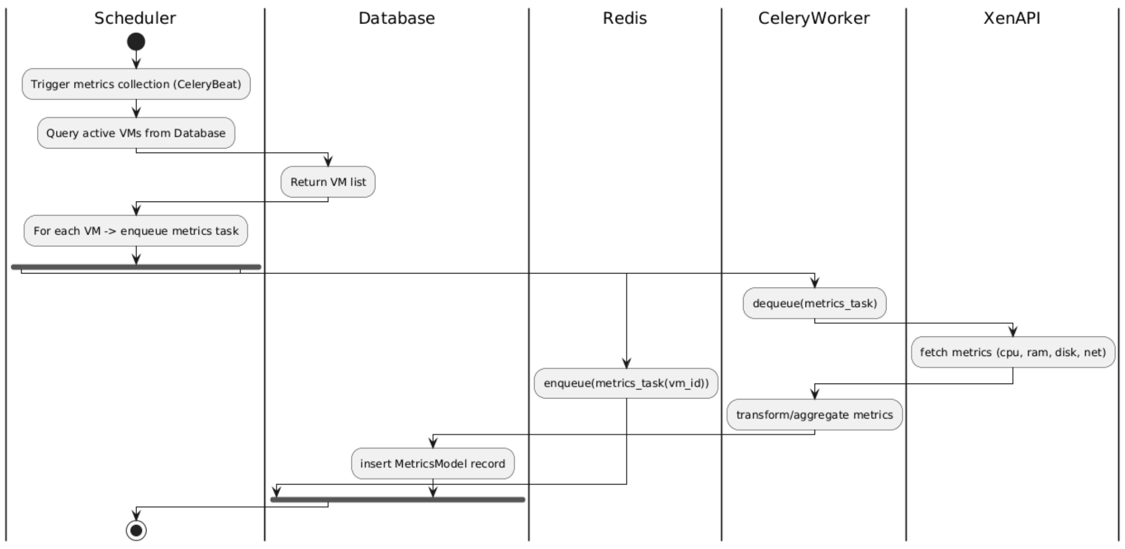

Scheduler
Database
Redis
CeleryWorker
XenAPI
Trigger metrics collection (CeleryBeat)
Query active VMs from Database
Return VM list
For each VM -> enqueue metrics task
dequeue(metrics_task)
fetch metrics (cpu, ram, disk, net)
enqueue(metrics_task(vm_id))
transform/aggregate metrics
insert MetricsModel record

Figure 4.13: Activity Diagram: Monitoring / Metrics Flow

Asynchronous Task Execution Flow

This activity diagram describes the lifecycle of any asynchronous task (create, migrate, delete, collect metrics) from enqueue to completion, including retries and error handling. Major steps:

- API enqueues a task in Redis
- Celery worker receives the task
- Execute action (with try/except)
- On success: update TaskLogModel, update VMModel/metrics as needed
- On failure: log error, apply retry policy, escalate if needed
- Notify caller (webhook/polling) with final status

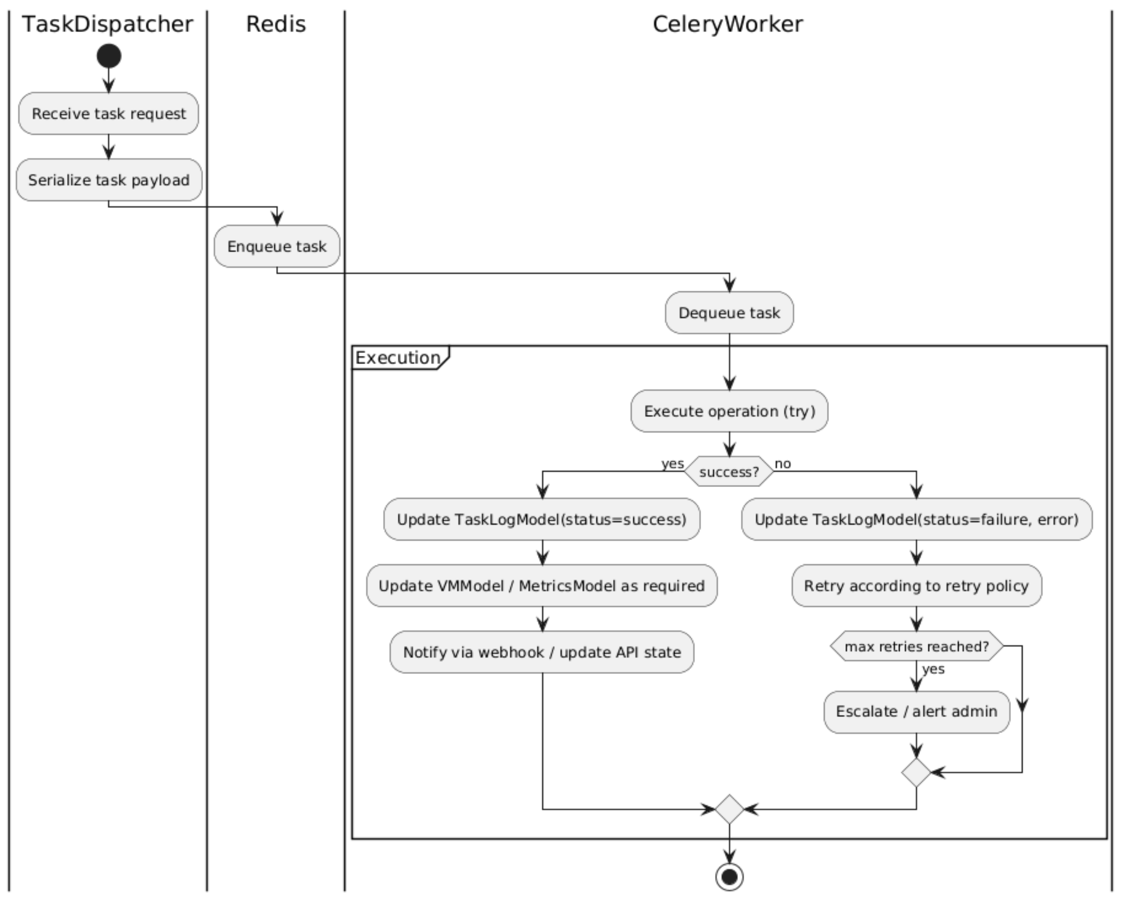


Figure 4.14: Activity Diagram: Asynchronous Task Execution Flow

4.3.3 State Diagrams

State diagrams (state machines) describe the possible life-cycle states of primary domain objects. For the Mini-Cloud Platform the most important object whose state needs modeling is the Virtual Machine (VM). The VM Lifecycle State Diagram captures valid states, allowed transitions, triggering events, and any transient states used during operations (e.g., migrating).

VM Lifecycle State Diagram

The VM lifecycle state diagram defines the complete lifecycle of a VM managed by the platform:

- **Template** – the VM is represented by a template/blueprint.
- **Provisioning** – worker is cloning/configuring VM resources.
- **Running** – VM is powered on and available.
- **Paused / Suspended** – VM is temporarily suspended.

- **Migrating** – live migration in progress (transient state).
- **Stopped** – VM is powered off.
- **Deleted** – VM has been removed; terminal state.

Transitions are triggered by API calls (create, stop, start, migrate, delete), worker completion events, or host-level failures. Retry and rollback transitions (e.g., provisioning failed → delete / cleanup) are also included.

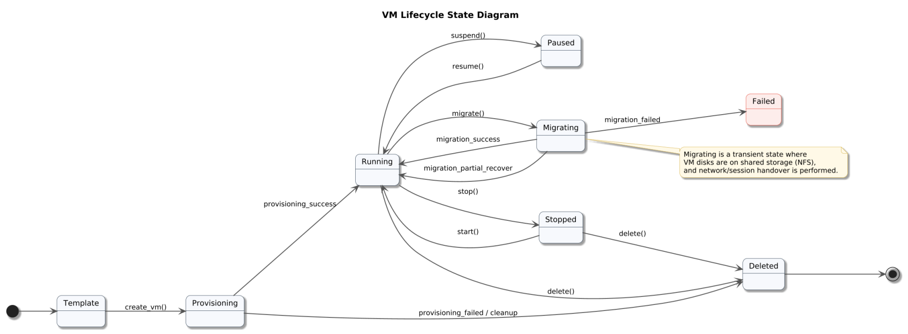

VM Lifecycle State Diagram
suspend()
Paused
resume()
Failed
migrate()
Migrating
migration_failed
migration_success
Running
migration_partial_recover
Migrating is a transient state where
VM disks are on shared storage (NFS),
and network/session handover is performed.
stop()
provisioning_success
Stopped
start()
delete()
Deleted
delete()
Template
create_vm()
Provisioning
provisioning_failed / cleanup

Figure 4.15: State Diagram: VM Lifecycle

4.4 Process Design & Flowcharts

The Process Design section illustrates the procedural workflows of major components in the Mini-Cloud Platform using flowcharts. These diagrams describe logical steps, decision points, and operational phases involved in VM creation, metrics collection, migration, and API-level request handling. Flowcharts provide a simplified visual representation of internal logic and help readers understand the high-level operational sequence without requiring knowledge of underlying implementation details.

4.4.1 VM Creation Flowchart

The VM Creation Flowchart represents the complete process executed when a user requests a new virtual machine. It includes validation, scheduling, task dispatching, provisioning, and final system updates.

Key steps include:

- Accept request from client
- Validate request payload
- Query scheduler for host selection

- Enqueue asynchronous VM creation task
- Celery worker provisions the VM
- Database is updated with VM metadata
- Response is returned to the user

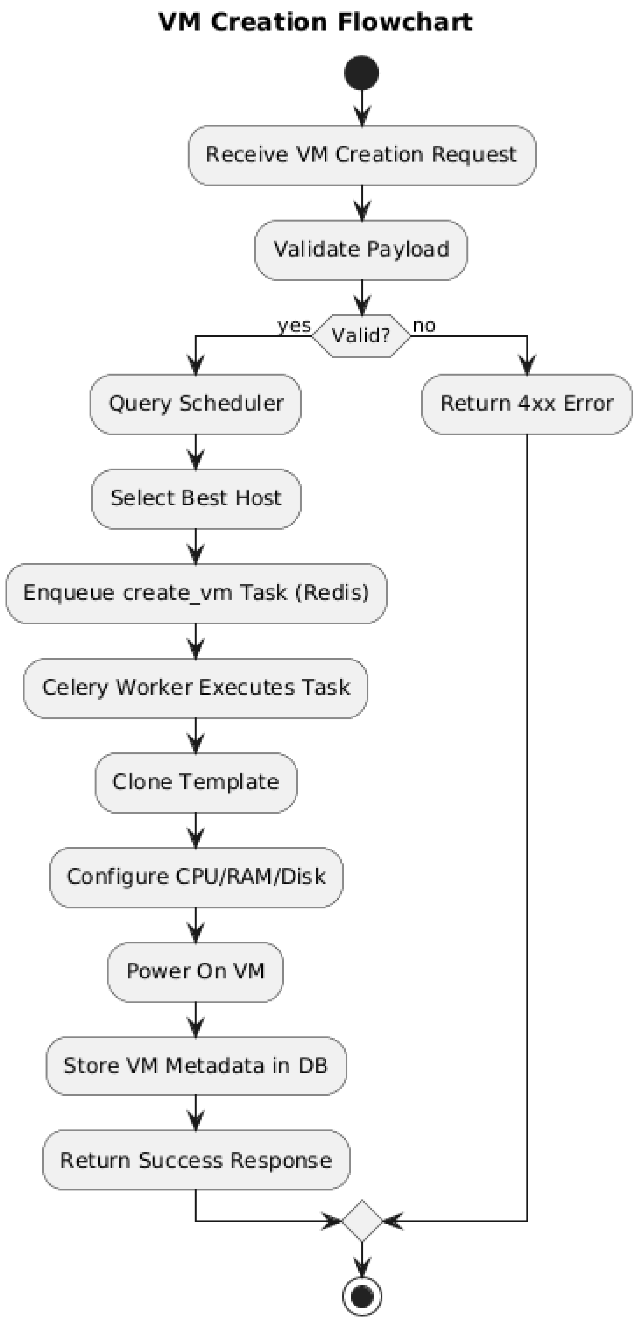


Figure 4.16: VM Creation Flowchart

4.4.2 Metrics Collection Flowchart

This flowchart describes the periodic collection of VM and host metrics used for monitoring and scheduling decisions. Metrics are gathered asynchronously and stored in the database. Key steps:

- Trigger collection via scheduler or Celery Beat
- Enqueue metrics collection tasks
- Fetch metrics from hypervisor via XOA
- Validate and format the collected metrics
- Store metrics into PostgreSQL

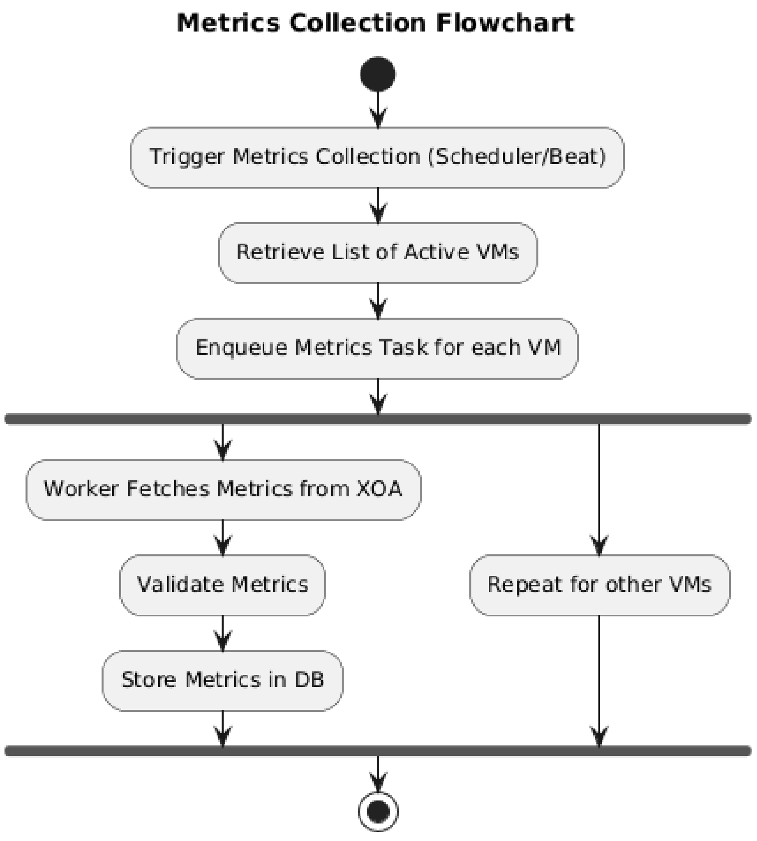

Metrics Collection Flowchart
Trigger Metrics Collection (Scheduler/Beat)
Retrieve List of Active VMs
Enqueue Metrics Task for each VM
Worker Fetches Metrics from XOA
Validate Metrics
Store Metrics in DB
Repeat for other VMs

Figure 4.17: Metrics Collection Flowchart

4.4.3 Migration Service Flowchart

The migration flowchart models the sequence followed when migrating a VM from one host to another. This process involves scheduler validation, asynchronous task execution, hypervisor interaction, and logging.

Steps include:

- Receive migration request
- Validate VM existence
- Request scheduler to verify target host
- Enqueue migration task into Redis
- Worker performs migration via XOA
- Update VM host entry in PostgreSQL
- Return migration status to the client

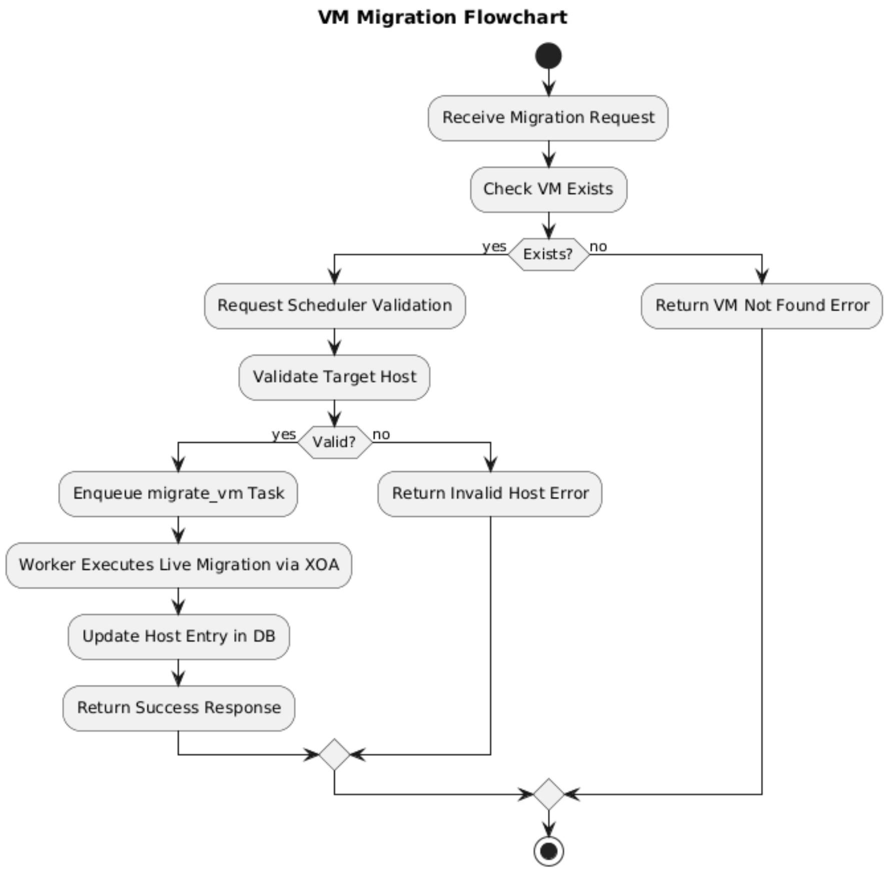


Figure 4.18: VM Migration Service Flowchart

4.4.4 API Request Processing Flow

This flowchart illustrates the processing pipeline for every incoming API request. It applies to VM creation, deletion, migration, and metrics retrieval endpoints.
The flow includes:

- Request reception
- Schema validation
- Routing to appropriate controller
- Scheduler interaction (if required)
- Enqueue tasks (if asynchronous)
- Generate response

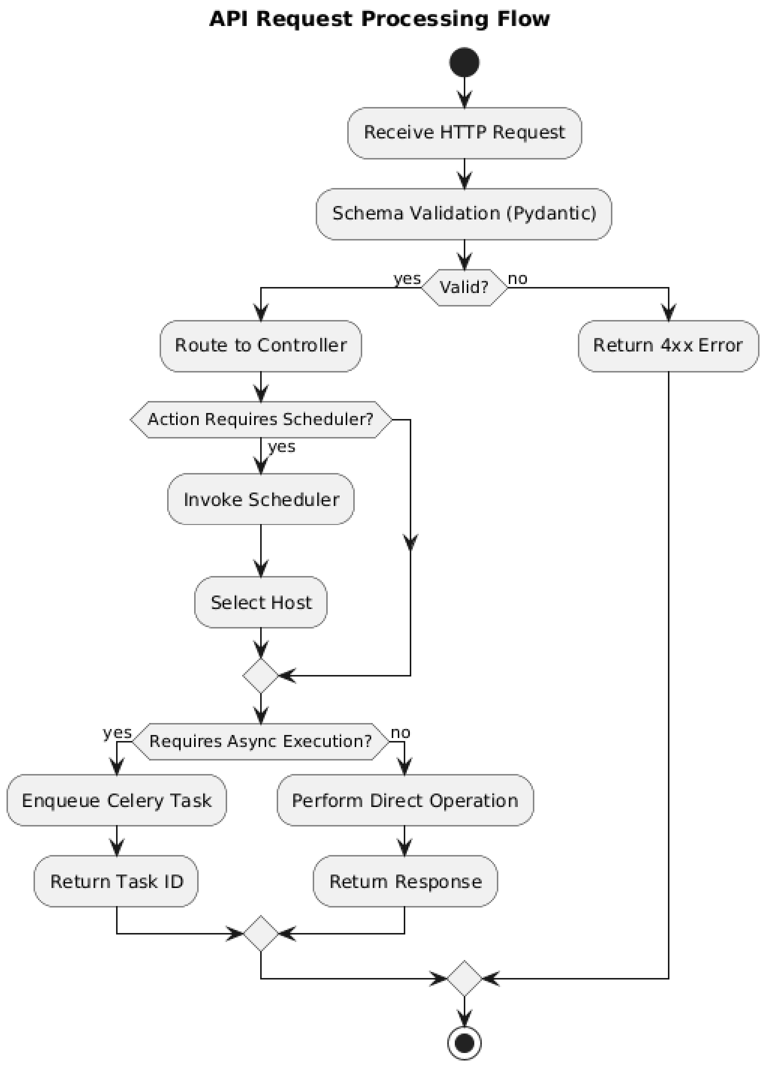


Figure 4.19: API Request Processing Flowchart

4.5 Algorithms and Pseudocode (Detailed)

This section presents detailed, production-quality pseudocode for the core algorithms used in the Mini-Cloud Platform. Each algorithm includes: purpose, inputs, outputs, preconditions, postconditions, main steps, error handling, and retry logic where applicable. The full, runnable source code for the implementation is included in Appendix A.

4.5.1 VM Creation Algorithm

Purpose: Provision a new virtual machine on the least-loaded hypervisor with cloud-init, ensure the VM boots, obtain its IP address, and persist metadata.

Inputs:

- `vm_name` – desired VM name
- `cpu`, `ram`, `disk` – resource requests
- `template` – template name or id
- `ssh_key` – public key path for cloud-init
- `preferred_host` (optional)

Outputs:

- `result` containing `vm_uuid`, `vm_ip`, `status`, `logs`

Preconditions:

- At least one host registered and reachable.
- Template exists on host or pool.
- SSH access to host with a key that can run required commands.

Postconditions:

- VM record is created in database if provisioning succeeds (best-effort).
- VM is started and IP is attempted to be detected.

Algorithm 1 VM Creation (detailed)

```
Require: vm_name, cpu, ram, disk, template, ssh_key, (preferred_host)
Ensure: result = {vm_uuid, vm_ip, status, logs}
 1: LOG("Start VM creation:", vm_name)
 2: host ← if preferred_host given then preferred_host else SelectLeastLoadedHost()
 3: if host == NULL then
 4:     LOG("ERROR: no available host"); RETURN error result
 5: end if
 6: workdir ← CreateTemporaryWorkdir(vm_name)
 7: seed_iso ← BuildCloudInitISO(workdir, ssh_key, vm_name)
 8: UploadISOToHost(seed_iso, host)
 9: iso_vdi ← EnsureISOImportedToSR(host, seed_iso)
10: if iso_vdi is NULL then
11:     LOG("WARN: ISO VDI not available; continuing with best-effort")
12: end if
13: template_uuid ← FindTemplateOnHost(host, template)
14: if template_uuid is NULL then
15:     LOG("ERROR: template not found on host"); RETURN error result
16: end if
17: vm_uuid ← CloneVMFromTemplate(host, template_uuid, vm_name)
18: if vm_uuid is NULL then
19:     LOG("ERROR: clone failed"); CLEANUP(workdir); RETURN error result
20: end if
21: ApplyResources(vm_uuid, cpu, ram, disk)
22: if iso_vdi ≠ NULL then
23:     AttachCD(vm_uuid, iso_vdi)
24: end if
25: StartVM(vm_uuid)
26: vm_ip ← DetectVMIPGuaranteed(host, vm_uuid)
27: if vm_ip == UNKNOWN then
28:     LOG("WARN: IP detection failed; marking IP as UNKNOWN")
29: end if
30: PersistVMRecordDB(vm_uuid, vm_name, host, cpu, ram, disk, vm_ip)
31: DetachAndDestroyCloudInitCD(vm_uuid)
32: CLEANUP(workdir)
33: LOG("VM creation finished:", vm_uuid, vm_ip)
34: return success result =0
```

IP Detection – summary of strategy (used in DetectVMIPGuaranteed):

1. Try xenstore keys such as `vm-data/ip` and `device/vif/0/ip`.
2. If not present, read domain id and try `/local/domain/<domid>/data/ip`.
3. Parse `vm-data/networks` for IPv4 tokens.
4. Use VIF MAC to consult ARP table, bridge FDB, neighbor table, and DHCP lease files on host.

5. Repeat with backoff up to a timeout (e.g., 2 minutes) and report `UNKNOWN` if not found.

4.5.2 Template Cloning Algorithm

Purpose: Clone an existing VM template into a new VM instance.

Inputs: `template_name`, `new_vm_name`, host.

Outputs: `vm_uuid` or error.

Algorithm 2 Template Cloning

```
Require: template_name, new_vm_name, host
 1: LOG("Find template on host")
 2: template_uuid ← FindTemplateOnHost(host, template_name)
 3: if template_uuid is NULL then
 4:     LOG("ERROR: template missing"); RETURN NULL
 5: end if
 6: vm_uuid ← RunCommandOnHost(host, "xe vm-clone uuid="+template_uuid+"
    new-name-label="+new_vm_name)
 7: if vm_uuid is NULL then
 8:     LOG("ERROR: clone failed"); RETURN NULL
 9: end if
10: return vm_uuid =0
```

4.5.3 Metrics Fetching Algorithm

Purpose: Periodically collect VM/host performance metrics and persist them.

Inputs: list of active VMs or hosts.

Outputs: persisted metrics records.

Algorithm 3 Metrics Fetching (parallel-safe)

```
Require: time_slice or triggered event
 1: vm_list ← QueryActiveVMsFromDB()
 2: for all vm IN vm_list in parallel do
 3:     metrics ← XenAPIClient.fetch_metrics(vm)
 4:     if metrics invalid then
 5:         LOG("WARN: metrics invalid for", vm)
 6:         continue
 7:     end if
 8:     NormalizeAndValidate(metrics)
 9:     InsertMetricsIntoDB(vm, metrics)
10: end for
11: LOG("Metrics collection completed for", |vm_list|) =0
```

Notes:

- Use worker concurrency (Celery pool) to parallelize metric retrieval across many VMs.
- Apply basic sanity checks on returned values and cap abnormal spikes before storing.

4.5.4 Celery Task Scheduling Algorithm

Purpose: Describe how tasks are scheduled, retried and executed via Celery and Redis.

Key features: enqueue, ack, retries, exponential backoff, result logging.

Algorithm 4 Celery Task Scheduling (high-level)

```
Require: task_definition (type, payload, retry_policy)
 1: TaskDispatcher → Serialize(task_definition)
 2: PushToRedisQueue(serialized)
 3: Worker ← WaitForTask()
 4: OnTaskReceived:
 5: ExecuteTaskHandler(payload)
 6: UpdateTaskLog(status=SUCCESS) TransientError
 7: If retries left: ScheduleRetryWithBackoff()
 8: UpdateTaskLog(status=RETRY) FatalError
 9: UpdateTaskLog(status=FAILED, error=msg)
10: RaiseAlertIfCritical() =0
```

4.5.5 VM Migration Algorithm

Purpose: Identify overloaded hosts and migrate appropriate VMs to balance load.

Inputs: host metrics history, VM list per host.

Outputs: migration actions invoked (or no-op).

Algorithm 5 VM Migration (detailed)

```
1: hosts ← GetAllHosts()
2: scored = []
3: for all host IN hosts do
4:    latest ← GetLatestMetric(host)
5:    if latest is NULL OR HostOverloaded(latest)=False then
6:       continue (or mark as candidate)
7:    end if
8:    score_info ← CalculateHostScore(latest) {uses cpu,mem,vmcount weights}
9:    Append(scored, (score_info.score, host))
10: end for
11: if len(scored) < 2 then
12:    LOG("Insufficient data to migrate"); RETURN no-action
13: end if
14: SortDescending(scored)
15: src = scored[0].host
16: dest = scored[-1].host
17: if scored[0].score - scored[-1].score < MIGRATION_THRESHOLD then
18:    LOG("Load difference below threshold"); RETURN no-action
19: end if
20: vm = SelectCandidateVMForMigration(src) {e.g., smallest VM by memory}
21: if vm is NULL then
22:    LOG("No candidate VM found"); RETURN no-action
23: end if
24: InvokeMigration(vm, dest) {xe vm-migrate or XOA API}
25: if migration succeeded then
26:    UpdateVMHostInDB(vm, dest)
27:    LogMigrationEvent(vm, src, dest)
28:    RETURN success
29: else
30:    LogMigrationFailure(vm, error)
31:    OptionalRollbackOrRetry()
32:    RETURN failure
33: end if=0
```

Notes:

- The migration algorithm should include safety checks (storage compatibility, network affinity).
- Prefer live migration with shared storage (NFS SR) to avoid VDI transfer.
- Conservative threshold (e.g., 0.15 score difference) reduces unnecessary migrations.

5. Hardware and Software Specifications

This section presents a comprehensive overview of the hardware and software ecosystem used to design, deploy, and evaluate the Mini-Cloud Virtualization Controller Platform. A cloud orchestration system depends on both its underlying compute infrastructure and its software environment; therefore, accurate specification of these components is essential for reproducibility, performance analysis, and system validation.

The Mini-Cloud Platform is implemented using two XCP-NG hypervisors, centrally managed using Xen-Orchestra (XOA), and controlled by a containerized FastAPI–Celery–Redis–PostgreSQL stack. This section describes each hardware and software component in detail.

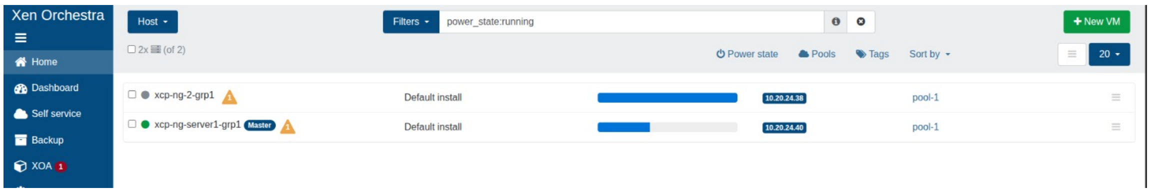
5.1 Hardware Setup

5.1.1 XCP-NG Servers

The virtualization layer is built using two physical servers running XCP-NG 8.3 as the bare-metal hypervisor. These servers are part of the same resource pool and support features such as VM provisioning, live migration, metrics extraction, and remote execution via XenAPI.

Table 5.1: Hypervisor Hardware Overview

Component	**Server 1**	**Server 2**
Model	HP EliteDesk 800 G1 SFF	HP EliteDesk 800 G1 SFF
CPU	Intel Core i7	Intel Core i7
RAM	16 GB DDR3	16 GB DDR3
Storage	500 GB SSD	500 GB SSD
NIC	Gigabit Ethernet	Gigabit Ethernet
Hypervisor OS	XCP-NG 8.3	XCP-NG 8.3
Management IP	10.20.24.40	10.20.24.38



power_state:running

xcp-ng-2-grp1 Default install 10.20.24.38 pool-1

xcp-ng-server1-grp1 Master Default install 10.20.24.40 pool-1

Figure 5.1: XCP-NG hypervisor nodes detected in Xen-Orchestra.

5.1.2 Host Specifications

Each host supports full hardware virtualization (Intel VT-x), Xen PVHVM guests, NFS-backed storage repositories, and hybrid resource scheduling. The hosts are configured with:

- Static IP addressing for reliability
- BIOS virtualization extensions enabled
- Shared NFS SR mounted for live migration
- XenAPI and SSH enabled for controller-driven provisioning

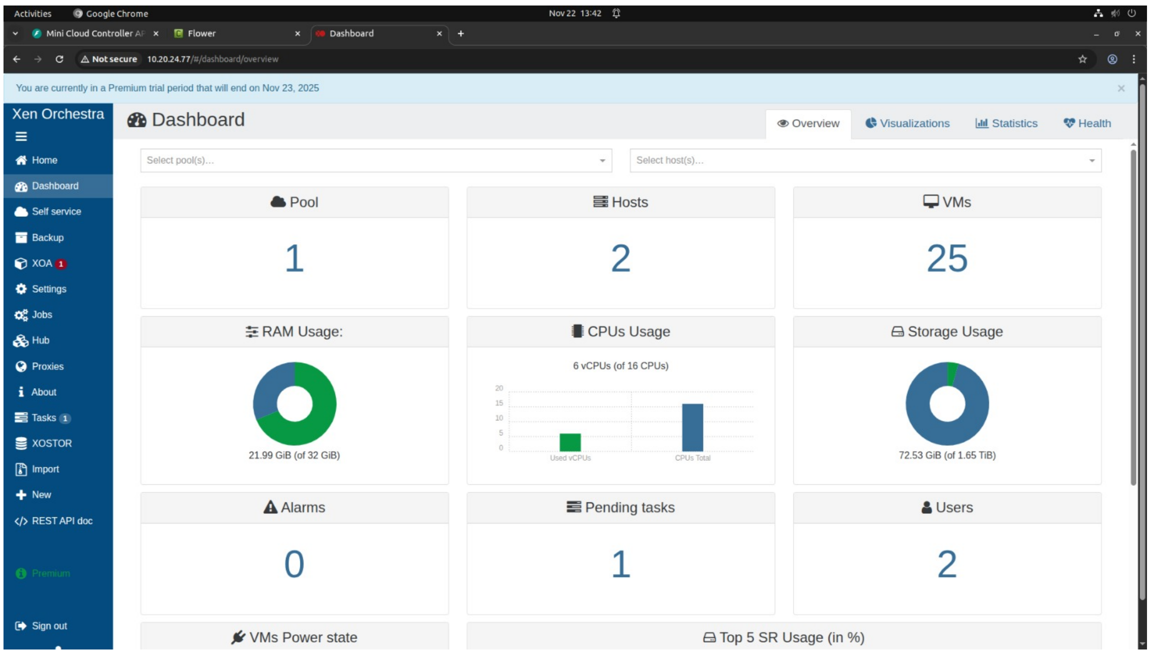


Figure 5.2: Xen-Orchestra dashboard showing RAM, CPU, storage, and pool-level statistics.

5.1.3 Network Topology

All components operate within a single Layer-2 switched network. The complete network configuration is:

- Subnet: `10.20.24.0/24`
- Gateway: `10.20.24.1`
- Controller Node: Docker-based FastAPI server

- Hypervisors: `10.20.24.40` and `10.20.24.38`
- Xen-Orchestra Virtual Appliance (XOA): `10.20.24.77`

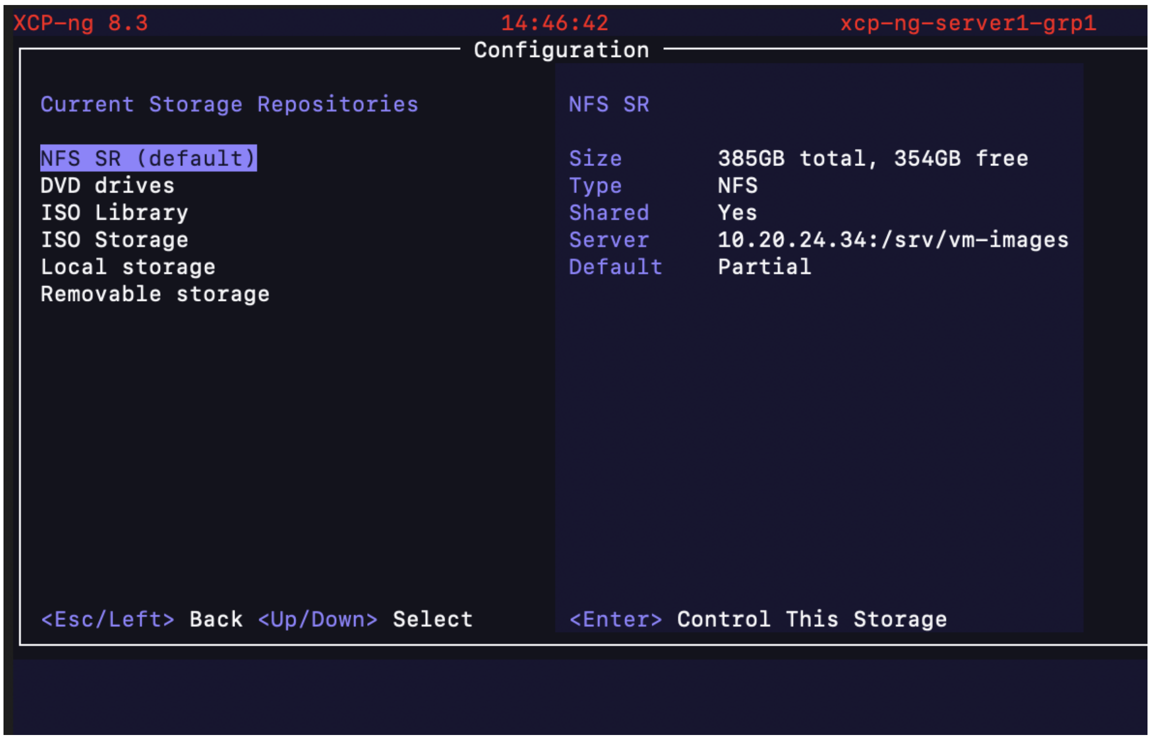
Four primary types of traffic are used:

1. API & provisioning traffic (controller → hypervisors)
2. Metrics reporting traffic (hypervisors → controller)
3. Storage traffic (hypervisors → NFS SR)
4. Migration traffic (hypervisor ↔ hypervisor)

5.1.4 Storage Environment

A shared NFS-based Storage Repository (SR) is used for all VM disk images and ISO resources. This configuration enables:

- VM live migration
- Centralized template management
- Cloud-init ISO storage
- Pool-wide VDI accessibility



```
XCP-ng 8.3                         14:46:42                    xcp-ng-server1-grp1
                                 Configuration

 Current Storage Repositories             NFS SR

 NFS SR (default)                         Size        385GB total, 354GB free
 DVD drives                               Type        NFS
 ISO Library                              Shared      Yes
 ISO Storage                              Server      10.20.24.34:/srv/vm-images
 Local storage                            Default     Partial
 Removable storage

 <Esc/Left> Back <Up/Down> Select         <Enter> Control This Storage
```

Figure 5.3: Shared NFS Storage Repository used by both XCP-NG hypervisors.

5.2 Software Stack

The controller platform is implemented using modern microservices and asynchronous task handling. This section describes the full software stack used.

5.2.1 Operating Systems

Host OS:

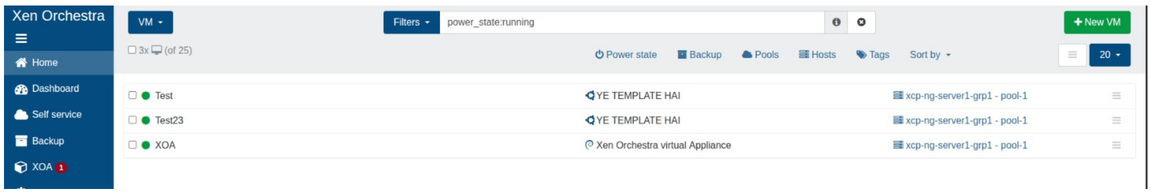
- XCP-NG 8.3 running directly on the physical hardware.

Controller OS:

- Ubuntu 22.04 LTS (Docker-based deployment)

Guest OS (VM Templates):

- Ubuntu 22.04 Cloud-init compatible images



power_state:running

3x (of 25)

● Test	YE TEMPLATE HAI	xcp-ng-server1-grp1 - pool-1
● Test23	YE TEMPLATE HAI	xcp-ng-server1-grp1 - pool-1
● XOA	Xen Orchestra virtual Appliance	xcp-ng-server1-grp1 - pool-1

Figure 5.4: Xen-Orchestra VM list showing active VMs and templates.

5.2.2 Python Version

All backend modules are written in **Python 3.11**. Core dependencies include:

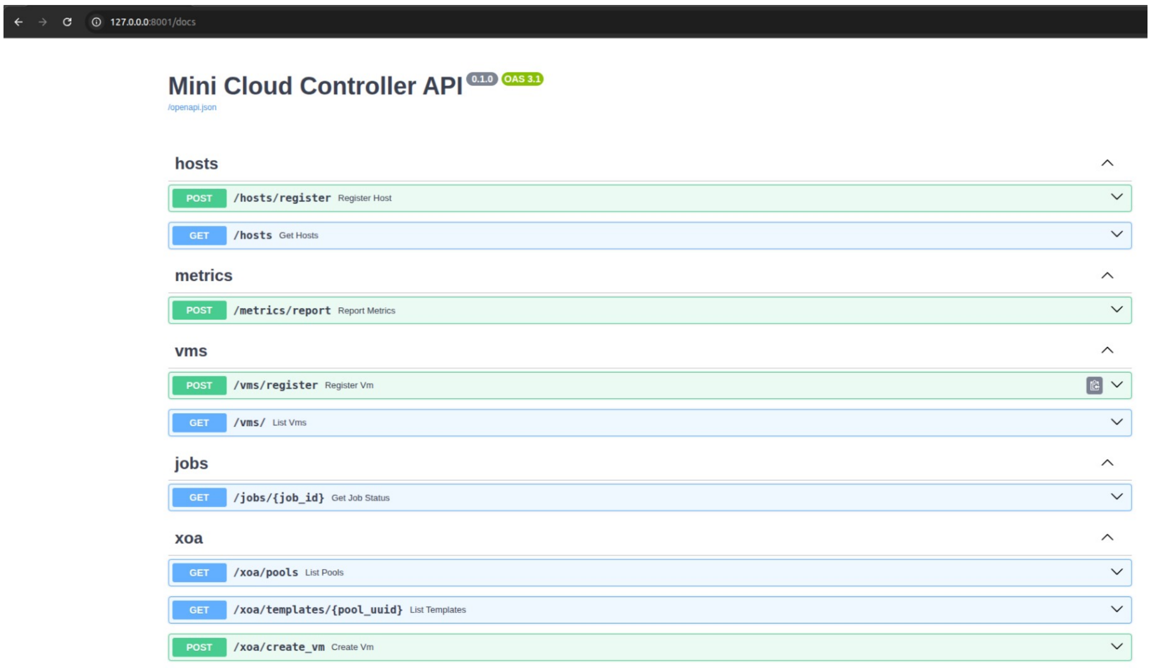
- `fastapi`
- `pydantic`
- `sqlalchemy`
- `celery`
- `redis`
- `requests`
- `uvicorn`

Python services are containerized for reproducibility and ease of deployment.

5.2.3 FastAPI, Celery, Redis, and PostgreSQL

FastAPI Serves as the main REST controller for:

- VM creation
- VM listing
- Host registration
- Metrics reporting
- Migration job dispatch



127.0.0.0:8001/docs

Mini Cloud Controller API 0.1.0 OAS 3.1

/openapi.json

hosts

POST `/hosts/register` Register Host

GET `/hosts` Get Hosts

metrics

POST `/metrics/report` Report Metrics

vms

POST `/vms/register` Register Vm

GET `/vms/` List Vms

jobs

GET `/jobs/{job_id}` Get Job Status

xoa

GET `/xoa/pools` List Pools

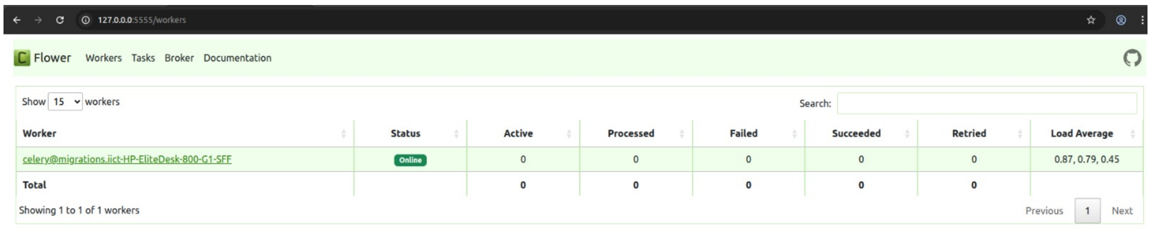
GET `/xoa/templates/{pool_uuid}` List Templates

POST `/xoa/create_vm` Create Vm

Figure 5.5: FastAPI OpenAPI documentation showcasing all exposed endpoints.

Celery Celery workers execute asynchronous jobs such as:

- VM provisioning
- Host metrics collection
- Migration triggers



127.0.0.0:5555/workers

Show 15 workers

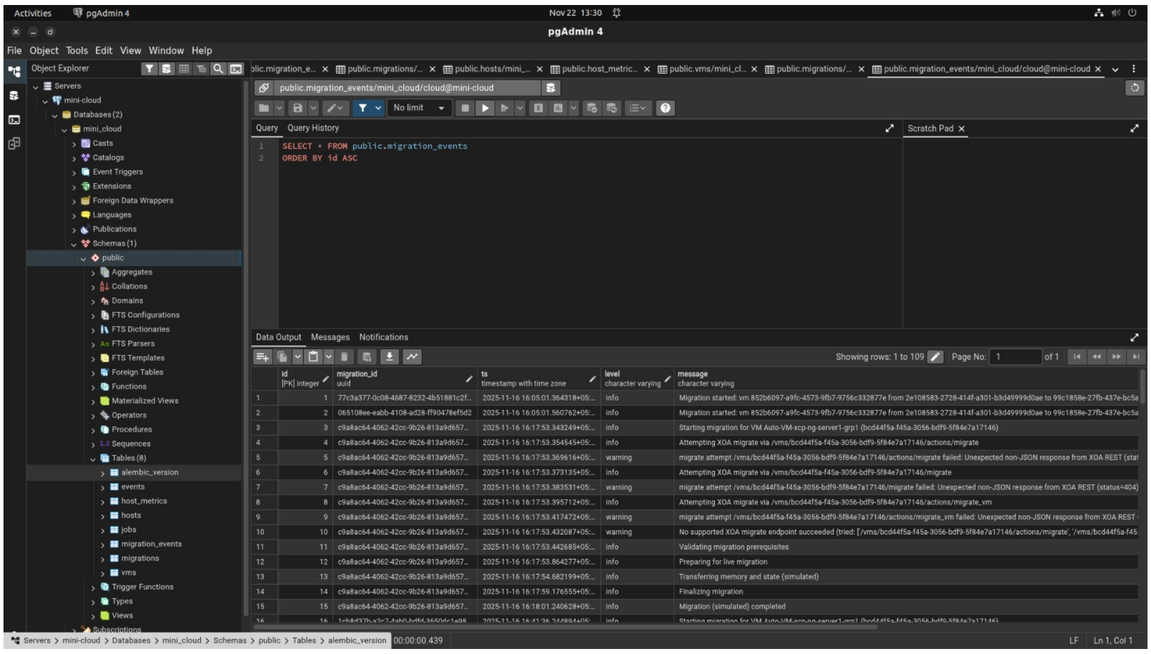
Worker	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@migrations.iict-HP-EliteDesk-800-G1-SFF	Online	0	0	0	0	0	0.87, 0.79, 0.45
Total		**0**	**0**	**0**	**0**	**0**	

Showing 1 to 1 of 1 workers

Figure 5.6: Celery Flower dashboard showing task queues and worker health.

Redis Used as the message broker for asynchronous task communication.

PostgreSQL Stores all relational data such as VM metadata, host metrics, migration events, and task logs.



public.migration_events/mini_cloud/cloud@mini-cloud

```
SELECT * FROM public.migration_events
ORDER BY id ASC
```

Showing rows: 1 to 109 Page No: 1 of 1

	id [PK] integer	migration_id uuid	ts timestamp with time zone	level character varying	message character varying
1	1	77c3a377-0c08-4687-8232-4b51881c2f...	2025-11-16 16:05:01.364318+05:...	info	Migration started: vm 852b6097-a9fc-4573-9fb7-9756c332877e from 2e108583-2728-414f-a301-b3d49999d0ae to 99c1858e-27fb-437e-bc5a
2	2	065108ee-eabb-4108-ad28-ff90478ef5d2	2025-11-16 16:05:01.560762+05:...	info	Migration started: vm 852b6097-a9fc-4573-9fb7-9756c332877e from 2e108583-2728-414f-a301-b3d49999d0ae to 99c1858e-27fb-437e-bc5a
3	3	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.343249+05:...	info	Starting migration for VM Auto-VM-xcp-ng-server1-grp1 (bcd44f5a-f45a-3056-bdf9-5f84e7a17146)
4	4	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.354545+05:...	info	Attempting XOA migrate via /vms/bcd44f5a-f45a-3056-bdf9-5f84e7a17146/actions/migrate
5	5	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.369616+05:...	warning	migrate attempt /vms/bcd44f5a-f45a-3056-bdf9-5f84e7a17146/actions/migrate failed: Unexpected non-JSON response from XOA REST (stat
6	6	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.373135+05:...	info	Attempting XOA migrate via /vms/bcd44f5a-f45a-3056-bdf9-5f84e7a17146/migrate
7	7	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.383531+05:...	warning	migrate attempt /vms/bcd44f5a-f45a-3056-bdf9-5f84e7a17146/migrate failed: Unexpected non-JSON response from XOA REST (status=404)
8	8	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.395712+05:...	info	Attempting XOA migrate via /vms/bcd44f5a-f45a-3056-bdf9-5f84e7a17146/actions/migrate_vm
9	9	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.417472+05:...	warning	migrate attempt /vms/bcd44f5a-f45a-3056-bdf9-5f84e7a17146/actions/migrate_vm failed: Unexpected non-JSON response from XOA REST
10	10	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.432087+05:...	warning	No supported XOA migrate endpoint succeeded (tried: ['/vms/bcd44f5a-f45a-3056-bdf9-5f84e7a17146/actions/migrate', '/vms/bcd44f5a-f45
11	11	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.442685+05:...	info	Validating migration prerequisites
12	12	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:53.864277+05:...	info	Preparing for live migration
13	13	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:54.682199+05:...	info	Transferring memory and state (simulated)
14	14	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:17:59.176555+05:...	info	Finalizing migration
15	15	c9a8ac64-4062-42cc-9b26-813a9d657...	2025-11-16 16:18:01.240628+05:...	info	Migration (simulated) completed
16	16	1cb8d37b-a2c7-4ab0-bdfd-3650dc1e98	2025-11-16 16:41:36.244894+05:	info	Starting migration for VM Auto-VM-xcp-ng-server1-grp1 (bcd44f5a-f45a-3056-bdf9-5f84e7a17146)

Servers > mini-cloud > Databases > mini_cloud > Schemas > public > Tables > alembic_version 00:00:00.439

Figure 5.7: pgAdmin view of metrics, events, and VM tables used by the controller.

5.2.4 Docker and Docker-Compose Setup

All backend services are deployed using Docker Compose. The major services include:

- `fastapi` service (REST controller)
- `celery_worker` service
- `celery_beat` scheduler
- `redis` message broker

- `postgres` database container

Containerization provides:

- Reproducibility
- Dependency isolation
- Fault isolation
- Simplified deployment

5.2.5 XOA / Xen-Orchestra Integration

Xen-Orchestra (XOA) acts as the management API for the hypervisor pool. It provides:

- Pool inspection (hosts, VMs, storage)
- Template listing
- REST API endpoints
- Live migration triggers

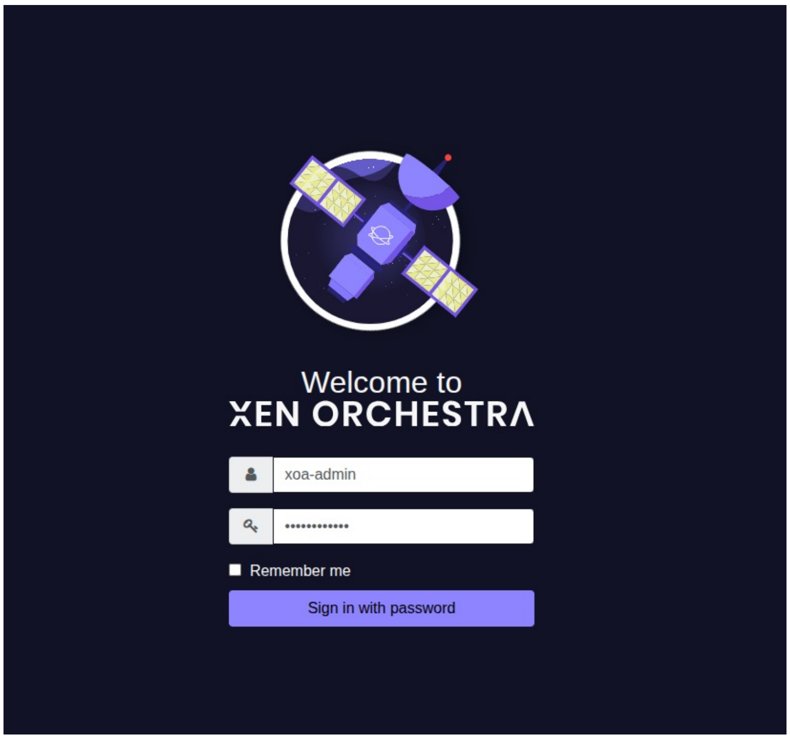


Figure 5.8: XOA Login interface used for cluster and resource orchestration.

5.2.6 Monitoring Tools

The Mini-Cloud Platform uses the following built-in monitoring capabilities:

- **Xen-Orchestra Dashboard**: Real-time tracking of CPU, RAM, storage, and VM activity.
- **Celery Flower Dashboard**: Monitoring asynchronous task execution.
- **pgAdmin**: Database query analysis and inspection.

5.2.7 Third-Party Libraries

Several external and system utilities support the controller's functionality:

- `genisoimage` (for cloud-init ISO generation)
- `openssh-client` (remote execution on hosts)
- `xe` CLI utilities (XenAPI interaction)
- `nfs-common` (shared storage mounting)

6. Cloud Services Categorization

Cloud computing services can be broadly divided into three main types of services: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). The Mini-Cloud Platform is a hybrid cloud orchestration based on the three-layers integration of components and as such, it is similar to the real-world cloud platforms like the AWS, Azure, or OpenStack.

This section involves a breakdown of the system components into the three cloud service models and how the component will be used to help the overall architecture.

6.1 Infrastructure as a Service (IaaS) Components

IaaS refers to provisioning raw compute, storage, and networking resources on demand. The following features in the Mini-Cloud Platform can be considered IaaS capabilities:

6.1.1 VM Provisioning

The system offers complete life cycle management of virtual machines as in AWS EC2, or Azure Virtual Machines. Through **FastAPI endpoints** and **Celery-based task execution**, the controller performs:

- Creation of VM on demand based on predefined templates.
- TM of VMs to the least-loaded hypervisor.
- CPU, memory, disk and network configuration.
- Exercise of provisioning scripts on XCP-NG hypervisors remotely.
- Live VM power operations (start, stop, delete).

VM provisioning pipeline resembles the normal IaaS operations in which raw compute instances are provided as services.

6.1.2 Template Management

Templates are pre-images of VM deployment. The Mini-Cloud Platform is compatible with:

- Listing templates that are available via XOA/XenAPI.

- Replicate templates after which new VM instances are created.
- Cloud-init based automation for SSH access and initialization.
- Managing template metadata (OS type, resource defaults).

These capabilities correspond directly to IaaS imaging and snapshot management services.

6.1.3 Network and Storage

It is a system that offers software-defined network and storage abstraction:

- XCP-NG virtual switch network bridging.
- NIC volume snapshoting upon creation of new VMs.
- Shared NFS-based storage repositories enabling:
  - Live migration
  - Pool-wide VDI sharing
  - Centralized ISO storage
- Scanning dynamic SRs in VM creation processes.

These features are in line with the essence of IaaS networking and storage services in commercial cloud systems.

6.2 Platform as a Service (PaaS) Components

PaaS decouples the infrastructure and offers an execution and development platform of applications. The Mini-Cloud Platform has a number of PaaS-like components to facilitate service orchestration, automation and hosting of application logic.

6.2.1 Celery Task Execution Platform

Celery acts as the distributed task execution layer of the platform, providing:

- Asynchronous and trustworthy job execution.
- Distributed processing across multiple workers.
- Built-in scheduling via Celery Beat.

- Decoupling of heavy operations (VM creation, metrics collection).
- Fault-tolerant job retries and status tracking.

This resembles managed PaaS offerings such as AWS SQS/Lambda integration or Azure Functions Task Queue systems.

6.2.2 API Management Layer

FastAPI provides the PaaS-level API gateway for:

- Validating requests using Pydantic schemas.
- Routing client requests to internal microservices.
- Integrating with XenAPI and Celery task pipelines.
- Managing API documentation via automatic OpenAPI generation.

This API layer functions similarly to AWS API Gateway or Google Cloud Endpoints.

6.2.3 Database Services

PostgreSQL is used as a backend service for data persistence. It provides structured storage for:

- VM metadata
- Host metrics
- Task logs
- Migration events
- Host registrations

Because the system exposes database services to higher-level components but not to end users, PostgreSQL operates as an internal PaaS subsystem.

6.3 Software as a Service (SaaS) Components

SaaS components represent fully managed, end-user accessible services requiring minimal configuration.

6.3.1 Xen-Orchestra Web Interface (SaaS Functionality)

Xen-Orchestra provides a browser-based, fully interactive management dashboard that includes:

- VM lifecycle control (start, stop, migrate, delete)
- Storage and pool monitoring
- Live charts for CPU, memory, and network usage
- Template browsing and import
- Hypervisor health monitoring

Although internally connected to our controller, XOA functions as a SaaS tool for administrators.

6.3.2 Custom API Services as SaaS Endpoints

The Mini-Cloud exposes a complete REST API service accessible to end users. These APIs provide:

- VM creation and deletion
- VMs, hosts and templates listing.
- Sending host metrics
- Triggering migration jobs
- Retrieving job statuses

Since these services are availed to end users (UI, CLI, or external programs), they serve as SaaS services.
REST API hence represents a SaaS layer over the PaaS backend logic and IaaS hypervisor resources.

7. Implementation

This part is a detailed summary of the actual application of the Mini-Cloud Platform. Its implementation includes the configuration of the virtualized infrastructure (XCP-NG and Xen-Orchestra), the deployment of the controller stack with the usage of Docker, and the creation of the REST API with FastAPI and the implementation of asynchronous tasks with the help of Celery. In each subsection, we shall detail the configuration options, reasons and details of how the system will be configured and deployed to run successfully.

7.1 Setting Up the Environment

The environment configuration consists of readying the hypervisor pool, configuring Xen-Orchestra (XOA), deploying the controller stack and making sure that all the services communicate. Two XCP-NG nodes, a centralized XOA appliance, and a controller host with the FastAPI service, Celery workers, Redis, and PostgreSQL are used in its implementation.

7.1.1 XCP-NG Pool Setup

The Mini-Cloud Platform is based on a cluster of XCP-NG hypervisor. The pool was configured in the following way:

- **XCP-NG 8.x** is installed on both physical servers.
- **Assigning static IPs**:
  - Node 1: `10.20.24.40`
  - Node 2: `10.20.24.38`
- **Setting up a pool**: The master host was created as the first host, and the second host was added to the master one to create a pool.
- **Shared Storage Configuration**: There was a NFS-based Storage Repository (SR) that was mounted on both the hosts. This enables:
  - Storage-level VM mobility
  - Live migration across nodes
  - Cloud-init image storage in ISO.

- **Template Import**: The Ubuntu cloud images were loaded into XOA and transformed into VM templates that can be used repeatedly.

Visual representations of the host pool, SR utilization, VM status pages and dashboard overview can be placed here to demonstrate the environment setup.

7.1.2 Connecting XOA with API Tokens

Xen-Orchestra (XOA) is the interface through which one can interact with the hypervisors. The Mini-Cloud Controller has the capability of VM provisioning, migration, and template management through XOA API based on the REST API.
The integration steps are:

1. Brower based logging into XOA.
2. Navigating to `Settings → Users → Tokens`.
3. Creating a new API token of the controller.
4. Storing the token in the controller's environment variables:

   ```
   XOA_URL="http://10.20.24.77"
   XOA_TOKEN="your_api_token_here"
   ```

5. Testing API connectivity using:

   ```
   curl -H "Authorization: Bearer $XOA_TOKEN" \
        http://10.20.24.77/api/pools
   ```

Once configured, the controller interacts with XOA for:

- Listing pools
- Retrieving templates
- Performing VM creation requests
- Executing live migration commands

7.1.3 Dockerizing the Controller

The Mini-Cloud Controller is packaged as a multi-container application using Docker Compose. The stack consists of:

- **FastAPI application container** — dealing with REST requests.
- **Container celery worker** — asynchronous job execution.
- **Redis container** — message broker
- **PostgreSQL container** — database backend
- **Celery Beat** — periodic source of metrics collection and migration

Key aspects of the deployment:

- **Dockerfile for the controller** defines dependencies and environment variables.
- **docker-compose.yml** orchestrates all containers.
- **Volume mapping** guarantees long term storage on databases.
- The definition of networks enables communication between containers.

The configuration allows to scale easily, reproduce and maintain. The Flower dashboard is exposed on port `5555` to monitor Celery tasks in real time.

7.2 FastAPI Implementation

FastAPI is the interface layer that is the main interface between the cloud system and the user. It offers organized API points, validation and routing of every cloud activity.
The application is of a modular structure:

- `routers/` — endpoint specifications
- `models/` — database ORM models
- `schemas/` — request/response schemas
- `tasks/` — Celery jobs
- `services/` — scheduler and business logic

7.2.1 API Endpoints

The core API groups include:

- **/hosts**
  - Register host
  - List available hosts
- **/metrics**
  - Post metrics from host agents
- **/vms**
  - Create VM (async)
  - List VMs
  - Delete VM
- **/jobs**
  - Track Celery job status
- **/xoa**
  - List pools
  - List templates
  - Trigger VM creation via XenAPI

Each endpoint uses Pydantic models for request validation and responds with structured data.

7.2.2 Authentication and Authorization

In the context of the given project, lightweight authentication is applied:

- Internal endpoints (metrics reporting, migration jobs) rely on trusted network access.
- The endpoints to users can use optional token-based authentication.
- Future integration options:
  - OAuth2 with JWT tokens
  - Rate limiting API gateway.

– Role-based access control

Although the prototype is designed with minimal security, the system architecture allows easily extending the authentication.

7.2.3 Models and Schemas

The data structure is characterized by two major layers:

1. **Pydantic Schemas** Schemas of input validation and output serialization Examples include:
   - `VMCreateRequest`
   - `HostRegistrationSchema`
   - `MetricsReportSchema`
2. **SQLAlchemy Models** Models Persistent in database. Examples include:
   - `VMModel`
   - `HostModel`
   - `MetricsModel`
   - `TaskLogModel`
   - `MigrationEvents`

FastAPI automatically integrates these schemas into an OpenAPI-compliant documentation page accessible at:

```
http://127.0.0.1:8001/docs
```

7.3 Celery + Redis Integration

The Mini-Cloud Platform is based on Celery and Redis to provide asynchronous and fault-tolerant and distributed tasks running. Redis is used as the messaging server, and Celery workers perform long term tasks as VM provisioning, live migration, and measuring. The integration will allow execution of heavy tasks asynchronously, scale up to multiple workers and offer powerful task monitoring.

7.3.1 Worker Tasks

Celery employees are specialized workers who have various tasks that are split into several categories:

- **VM Lifecycle Tasks:** VM creation, VM deletion and post-provisioning.
- **Metrics Tasks:** Periodically collect host metrics from XCP-NG using XOA APIs.
- **Migration Tasks:** Automatic migration and live migration of VMs in case a host is overloaded.
- **Utility Tasks:** Update logs, refresh templates, or perform periodic validations.

Every operation is run independently and the failures can be re-run without affecting the response of the system. The Flower dashboard is used to monitor the worker logs and real time statistics.

7.3.2 Scheduling Tasks

Celery Beat is the internal scheduler that will be the trigger of periodic functions:

- **Record measurements after every 2 minutes**
- **Measure host load and cause migrations after every 2 minutes**

The scheduling configuration is defined directly inside the Celery application:

- Assures calculation of new metrics by every node.
- Maintains synchronization between worker nodes and controller state.
- Automates load balancing and scaling decisions.

This scheduling system automates the behavior of clouds like a commercial system like VMware DRS or OpenStack Watcher.

7.3.3 Error Handling

Celery provides robust error-handling mechanisms:

- **Automatic Retries:** Tasks retried at preset delays in case of unavailability of a resource or a network.

- **Exception Logging:** The errors are then logged into the task logs table.
- **Partial Failure Recovery:** VM creation failures will provide detailed logs which can be used to debug provisioning problems.
- **Worker Isolation:** When a task fails, it does not impact on the functioning of a worker or other tasks.

This will provide consistency in queueing of tasks even when a network or a hypervisor is unstable

7.4 PostgreSQL Database Implementation

PostgreSQL is the persistent storage of all metadata, operational logs and time-series metrics. The database structure ensures referential integrity and supports such analytical procedures like historical load tracking, migration auditing.

7.4.1 Schema Design

The database design is based on a normalized relational design whose key tables include:

- **VM Table:** Metadata on all VMs created such as UUID, assigned host, CPU, RAM and time of creation.
- **Host Table:** Maintains registered hypervisors and their identifiers.
- **Table of metrics:** Host-by-host performance.
- **Task Logs Table:** Records Celery task status, errors, and timestamps.
- **Migration Events Table:** Records the decision, the outcome, and the time of every migration attempt.

The schema is made to accommodate:

- Effective queries on metrics.
- Historical audits.
- Foreign key data integrity.
- Additional features like autoscaling or notification can be added in the future.

7.4.2 Session Management

Database access uses SQLAlchemy ORM with dependency-injected sessions:

- A New session is generated whenever an API request is received.
- Based on the outcomes, sessions are rolled back or committed accordingly.
- Independent sessions are used to prevent interference with background tasks (Celery).
- Connection pooling provides low-latency requests of high-frequency metrics.

Safety The session scopes are separated in a clean way that allows the parallel execution of the API and Celery workers.

7.5 Complete VM Creation Pipeline

The most complicated process in the system is the VM creation process, which implies the coordination of the API, scheduler, controller scripts, and the hypervisor actions through XCP-NG CLI tools.
The following stages are the major ones in the pipeline:

7.5.1 Using SSH and xe CLI

VM provisioning is executed remotely on the selected hypervisor using:

- `ssh` for remote command execution
- `scp` for transferring cloud-init ISO files
- `xe` CLI commands for template cloning, disk creation, network attachment, and VM power operations

Such an approach is consistent with official XCP-NG automation processes and does not require scripts to be installed within XOA.

7.5.2 Template UUID Handling

The identification of templates is made based on their distinct UUIDs which are accessed by:

```
xe template-list name-label="UbuntuFix"
```

The scheduler gets the correct template depending on:

- User request
- OS type
- Template-to-host compatibility

Template cloning creates a VM, which has inherited properties.

7.5.3 IP Assignment

Cloud-init injects SSH keys and generates metadata needed for VM initialization. The VM's IP address is resolved using a multi-step XenStore lookup:

1. `xenstore-read vm-data/ip`
2. `xenstore-read device/vif/0/ip`
3. Searching DHCP leases and bridge FDB tables
4. ARP-table-based fallback scanning

The system makes 60 tries to recover the IP, which is reliable and, therefore, IP detection is strong when the OS vary

7.5.4 Attaching Network

The network interfaces are connected with:

```
xe vif-create vm-uuid=<vm> network-uuid=<net> device=0
```

This ensures that:

- VMs are given proper VLAN or bridge mappings.
- There is consistency in connectivity between hypervisors.
- Network routing is not disturbed by migration.

7.6 Metrics Collection Service

Constant metrics based on metrics collection, allows smart scheduling and automation of migration decisions. It is based on the metrics API of XOA to fetch real-time data of hypervisors.

7.6.1 Using XOA Metrics API

The metrics obtained are stored as the host metrics table with:

- CPU usage percentage
- Memory usage percentage
- Count of running VMs
- Network I/O (if enabled)

Each call uses the XOA token set in environment variables and is executed inside a Celery task.

7.6.2 Storing Metrics in DB

The metrics obtained are stored as the host metrics table with:

- Host ID
- CPU percentage
- Memory percentage
- VM count
- Timestamp

The scheduler then uses these values to calculate normalized scores of load.

7.6.3 Dashboard Options

The system facilitates other visualization:

- XOA pool, host and VM usage dashboard.
- Grafana or Prometheus (future enhancement).
- Custom FastAPI endpoints to query historical metrics.

Such dashboards give details on cluster health and help in both manual and automated VM lifecycle decisions.

7.7 VM Migration Service

A VM Migration Service that is going to redistribute the virtual machines on the XCP-NG cluster in an intelligent way and distribution is also in charge. The service constantly checks the metrics of the host, calculates load scores, and carries out live migrations with Xen-Orchestra (XOA) APIs and the underlying xen hypervisor with the xe command-line tools.

The migration service works independently on Celery tasks and provides the platform with load balance, bottlenecks, and real-time response to overload.

7.7.1 Design

The migration subsystem will have three significant building blocks:

1. **Metrics Collector** Every minute CPU, memory and VM-count data of every host is collected and saved into the database.

2. **Scheduler (Decision Engine)** Calculates normalized load scores of each host with:

$$\text{score} = (0.5 \cdot \text{cpu\_norm}) + (0.3 \cdot \text{mem\_norm}) + (0.2 \cdot \text{vmc\_norm})$$

   The host whose score is greatest is taken to be the most loaded and that which scores least is the least loaded.

   When the difference between scores is more than some specific threshold (e.g., 0.15), a migration is evoked.

3. **Migration Executor** In charge of:

   - It is to choose the smallest VM on the overloaded host.
   - Finding a target host that is partially loaded.
   - Carrying out the live migration by XOA or xe vm-migrate.

The system allows manual and automated migrations. The automated migration pipeline is triggered every 2 minutes by Celery Beat.

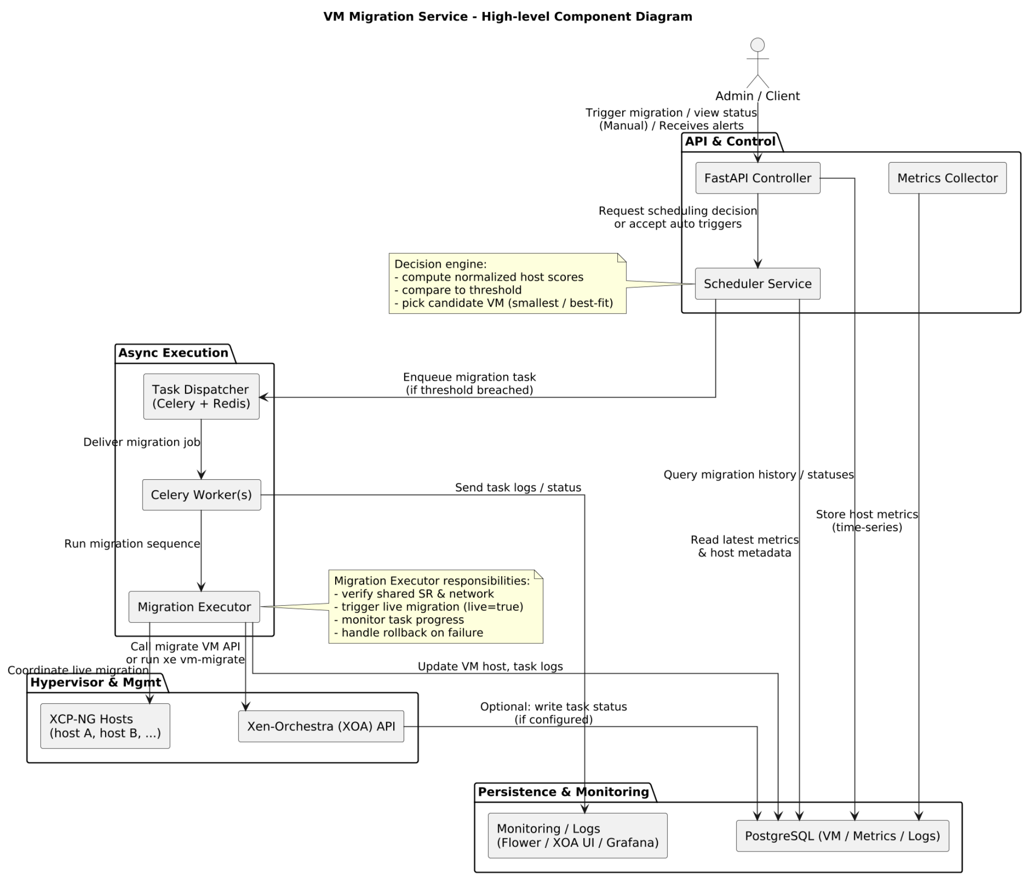

VM Migration Service - High-level Component Diagram
Admin / Client
Trigger migration / view status (Manual) / Receives alerts
API & Control
FastAPI Controller
Metrics Collector
Request scheduling decision or accept auto triggers
Decision engine:
- compute normalized host scores
- compare to threshold
- pick candidate VM (smallest / best-fit)
Scheduler Service
Async Execution
Task Dispatcher (Celery + Redis)
Enqueue migration task (if threshold breached)
Deliver migration job
Celery Worker(s)
Send task logs / status
Query migration history / statuses
Store host metrics (time-series)
Read latest metrics & host metadata
Run migration sequence
Migration Executor
Migration Executor responsibilities:
- verify shared SR & network
- trigger live migration (live=true)
- monitor task progress
- handle rollback on failure
Call migrate VM API or run xe vm-migrate
Coordinate live migration
Update VM host, task logs
Hypervisor & Mgmt
XCP-NG Hosts (host A, host B, ...)
Xen-Orchestra (XOA) API
Optional: write task status (if configured)
Persistence & Monitoring
Monitoring / Logs (Flower / XOA UI / Grafana)
PostgreSQL (VM / Metrics / Logs)

Figure 7.1: High-level design of the VM Migration Service.

7.7.2 Payload Format

The migration request—whether issued manually via API or internally through Celery—uses a well-defined JSON payload format. The payload will include some information concerning the VM to migrate, the target host, and optional flags.

```
{
    "vm_uuid": "f2b87c3c-345b-67e8-e8ea-51da8001fadc",
    "target_host_uuid": "656b8fc2-fed7-5820-f797-a1e98b7b2e74",
    "live": true
}
```

Field descriptions:

- `vm_uuid`: Unique identifier of the VM to be migrated.
- `target_host_uuid`: The Xen host where the VM should be placed.
- `live`: Boolean flag indicating whether a live (zero-downtime) migration is required.

These values are automatically generated by the scheduler in case of an automatic migration. In a manual request when one uses FastAPI, the user would provide the VM UUID or the VM name.

7.7.3 XOA Migration API

The platform has an integration with the REST API of Xen-Orchestra to programmatically migrate VMs. The following API call is the migration workflow through XOA:

```
POST /vms/{vm_uuid}/migrate
Authorization: Bearer <XOA_TOKEN>
Body:
{
    "destination": "<host_uuid>",
    "live": true
}
```

Advantages of using XOA API:

- Handles both live and cold migrations.
- Automatically validates:
  - shared availability of storage,
  - network compatibility,
  - host availability.
- Gives regular task logs and status reports.
- Removes the possibility of having to hand over storage or network path.

Task progress is also revealed by the XOA internal task management system that can be queried or logged by the controller.

7.7.4 Cross-Host Migration

The cross-host migration is made possible with::

- **Shared NFS Storage Repository** Provides access to VM disks (VDIs) in both hosts at the same time.
- **Network Consistency** Hypervisors are bridged with the same network so that there is smooth NIC reattachments in the migration process.

- **Parameters of Live Migration Run with:**

  ```
  xe vm-migrate vm=<vm_uuid> host=<target_uuid> live=true
  ```

- **Scheduler Logic** Ensures:
  - The source host is also overloaded.
  - The attacker possesses sufficient CPU and memory on the target host.
  - The least VM is migrated in order to reduce interference.

Live migration has a zero-downtime operation. During migration:

- The VM RAM is sent to the host.
- Youngest network sessions are still open.
- CPU state is transferred atomically.

Once migration completes:

- In PostgreSQL, VM metadata is updated.
- The migration is recorded by Celery.
- The host load scores are recalculated by the scheduler.

This forms a dynamically balanced self-regulating system of workload, allowing the absence of involvement with the user.

8. Sample Source Code

This section presents representative excerpts of the most important components of the Mini-Cloud Platform. Only essential and illustrative code segments are included here to explain the system's structure and behavior.

The complete and fully annotated source code is provided in **Appendix A**.

8.1 FastAPI API Endpoints

The FastAPI layer provides REST endpoints for VM lifecycle operations. Below is an excerpt of the primary VM creation endpoint:

Listing 8.1: FastAPI VM Creation Endpoint

```python
@router.post("/create")
def create_vm(payload: VMCreateRequest, db: Session = Depends(
    get_db)):
    TEMPLATE_NAME = payload.template or "Ubuntu-24.04-Base"
    POOL_ID = "b4bd9096-7d00-aeea-c35c-16bfba69b778"

    try:
        result = schedule_vm_custom(
            db=db,
            user_cfg=payload.dict(),
            template_name=TEMPLATE_NAME,
            pool_id=POOL_ID
        )
        return {"status": "success", "details": result}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

This endpoint validates user input, selects an appropriate hypervisor, and dispatches the VM creation process through Celery.

8.2 Celery Task Code

Celery workers handle asynchronous operations such as VM creation, metrics collection, and migration checks. The following snippet shows the metric collection job:

Listing 8.2: Celery Metric Collection Job

```python
@celery_app.task(name="app.tasks.jobs.collect_metrics_job")
def collect_metrics_job():
    try:
        collect_metrics()
        return {"status": "success"}
    except Exception as e:
        logger.exception("collect_metrics_job failed")
        return {"status": "error", "error": str(e)}
```

Periodic scheduling is performed through Celery Beat.

8.3 VM Creation Script (Excerpt)

The VM provisioning logic is executed via an external Bash script on the target XCP-NG host. Below is a representative excerpt demonstrating template cloning and VM startup:

Listing 8.3: Excerpt from VM Creation Script

```bash
# clone template -> VM
VM_UUID=$(remote "xe vm-clone uuid=$TEMPLATE_UUID \
    new-name-label=\"$VM_NAME\" --minimal")

# set CPU and RAM limits
remote_quiet "xe vm-param-set uuid=$VM_UUID \
    VCPUs-max=$VCPUS VCPUs-at-startup=$VCPUS"
remote_quiet "xe vm-memory-limits-set \
    static-min=${RAM_MB}MiB static-max=${RAM_MB}MiB \
    dynamic-min=${RAM_MB}MiB dynamic-max=${RAM_MB}MiB"

# start VM
remote_quiet "xe vm-start uuid=$VM_UUID"
echo "[INFO] VM started successfully"
```

The full script (300+ lines) is included in Appendix A.

8.4 Metrics Fetching Code

Host load is computed through a simple but effective scoring algorithm. The following snippet shows the normalization logic:

Listing 8.4: Host Scoring Function

```python
def calculate_host_score(metric):
    cpu_norm = float(metric.cpu_percent) / 100.0
    mem_norm = float(metric.mem_percent) / 100.0
    vmc_norm = min(metric.vms_running / 10.0, 1.0)

    score = (cpu_norm * 0.5) + \
            (mem_norm * 0.3) + \
            (vmc_norm * 0.2)

    return {"cpu_norm": cpu_norm,
            "mem_norm": mem_norm,
            "vmc_norm": vmc_norm,
            "score": score}
```

This scoring function is used by the scheduler to detect overloaded hosts.

8.5 Migration Code (Excerpt)

The final migration logic determines overloaded hosts, selects a VM, and triggers `xe vm-migrate`.

Listing 8.5: Excerpt from Migration Engine

```python
if overloaded_score - under_score >= 0.15:
    vm = db.query(models.VM)\
        .filter(models.VM.host_id == src.id)\
        .order_by(models.VM.memory_mb.asc())\
        .first()

    cmd = f"xe vm-migrate vm={vm.uuid} \
             host={dest.uuid_xen} live=true"
    subprocess.run(shlex.split(cmd), check=True)
```

Only the core logic is shown here; full implementation in Appendix A contains extensive logging and error handling.

8.6 Docker Compose File (Excerpt)

The controller stack is deployed through Docker Compose. The following snippet highlights the main services:

Listing 8.6: Excerpt from docker-compose.yml

```yaml
services:
  api:
    build: .
    ports:
      - "8001:8001"
    depends_on:
      - redis
      - postgres

  celery_worker:
    build: .
    command: celery -A app.tasks.celery_app worker --loglevel=
       INFO
    depends_on:
      - redis
      - postgres

  redis:
    image: redis:latest

  postgres:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: example
```

8.7 Database Models (Excerpt)

Database structures are defined using SQLAlchemy ORM. Below is an excerpt of the VM and Host models:

Listing 8.7: Excerpt from Database Models

```
class Host(Base):
    __tablename__ = "hosts"
    id = Column(Integer, primary_key=True)
    name = Column(String, unique=True, nullable=False)
    ip = Column(String, nullable=True)

class VM(Base):
    __tablename__ = "vms"
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    uuid = Column(String, unique=True)
    host_id = Column(Integer, ForeignKey("hosts.id"))
```

Additional models for metrics and task logs appear in Appendix A.

9. Results and Discussions

This chapter presents the experimental outcomes of the Mini-Cloud Platform, along with an analytical discussion of system behaviour, performance characteristics, automation accuracy, and comparison with existing tools. Outputs include real logs, performance metrics, API responses, Celery task traces, VM lifecycle results, and migration evaluation. All observations are based on live execution on a two-node XCP-NG pool connected to Xen-Orchestra (XOA) and controlled by the FastAPI–Celery–PostgreSQL–Redis orchestration pipeline.

9.1 Output Screenshots and System Logs

This section provides the actual logs generated by the system during VM provisioning, periodic metrics collection, task execution, and migration evaluation. These results validate the correctness, automation efficiency, and production-like behaviour of the developed platform.

9.1.1 XOA VM Creation Output

Upon invoking the API endpoint `/vm/create`, the scheduler selects the least-loaded hypervisor using the weighted scoring model, and the VM creation script executes over SSH. The following is the real log captured during a successful VM creation:

Listing 9.1: VM Creation Log from XOA and Controller

```
==================== SCHEDULER START ====================
[SCHEDULER][HOST: xcp-ng-server1-grp1]
CPU_raw: 1.06%      cpu_norm=0.011
MEM_raw: 76.25%      mem_norm=0.762
VM_count_raw: 0      vmc_norm=0.000
FINAL SCORE: 0.23353

[SCHEDULER][HOST: xcp-ng-2-grp1]
CPU_raw: 1.13%      cpu_norm=0.011
MEM_raw: 76.27%      mem_norm=0.763
VM_count_raw: 0      vmc_norm=0.000
FINAL SCORE: 0.23390

[SELECTED HOST]
xcp-ng-server1-grp1
```

```
(UUID=2e108583-2728-414f-a301-b3d49999d0ae, IP=10.20.24.40)
==================== SCHEDULER END ====================

[Scheduler] Selected host:
xcp-ng-server1-grp1 (2e108583-2728-414f-a301-b3d49999d0ae)
ip=10.20.24.40

[Scheduler] EXECUTING:
/app/app/create_vm_remote_fixed.sh --host 10.20.24.40 --name
    ShlokPlease
--template Ubuntu-24.04-Base --cpu 1 --ram 2048 --disk 20GiB
--network 'Pool-wide network associated with eth0' --ssh-key /
    root/.ssh/id_rsa.pub

[INFO] Running VM creation on host: 10.20.24.40
[INFO] Looking up template 'Ubuntu-24.04-Base'
[INFO] Template UUID: ce9a2aba-ba83-2ed8-b3f4-6211d3e7db2e
[INFO] VM UUID: 34ee19ab-16a9-9717-5bfe-a2cf710763a2
[INFO] Template uses UEFI; setting VM to UEFI
[INFO] Resizing VDI ...
[INFO] Importing ISO...
[INFO] Starting VM 34ee19ab-16a9-9717-5bfe-a2cf710763a2 ...

--------------------------------------------
VM CREATED SUCCESSFULLY
Name : ShlokPlease
UUID : 34ee19ab-16a9-9717-5bfe-a2cf710763a2
IP : 10.20.24.135
ISO VDI (attached/imported) : 7371338b-0623-4354-a8ad-
    d410046ef585
--------------------------------------------
```

This log confirms:

- Correct host selection by scheduler.
- Successful template cloning.
- Resource configuration performed correctly.
- Cloud-init ISO generation and upload succeeded.
- Non-disruptive CD detachment after boot.

- Real-time IP detection worked through Xenstore and ARP fallbacks.

9.1.2 Controller Logs

The controller logs record all Celery, scheduler, API, and DB events. Below is a real section of controller output:

Listing 9.2: Controller Log for VM Creation

```
INFO: 172.19.0.1:40436 - "POST /api/vm/create HTTP/1.1" 200 OK

[Scheduler] VM Created     UUID = 7371338b-0623-4354-a8ad-d410046ef585
[Scheduler] IP extracted from script     10.20.24.135
```

These logs verify:

- End-to-end path from POST request to successful VM provisioning.
- Database insertion of VM metadata.
- Proper return of VM UUID and IP in API response.

9.1.3 Worker Task Completion Logs

Celery workers execute asynchronous jobs such as VM creation, metrics fetching, and migration evaluation. Below are actual output logs for periodic metrics collection:

Listing 9.3: Worker Log for Metrics Collection

```
[2025-11-18 15:30:24]  Scheduler     Triggered task: collect-host-metrics-every-2-minutes
[15:30:24.110]  Task received     app.tasks.jobs.collect_metrics_job
                                [9ce713ea-eb81-4b4a-8ad1-ac13bf2d2f17]

[15:30:24.110]  COLLECTOR Running periodic host metrics collection...

[15:30:24.132]  Found 2 hosts via XOA REST API
[15:30:24.153]  Updated host     xcp-ng-server1-grp1
[15:30:24.223]  Metrics saved for xcp-ng-server1-grp1
                CPU usage : 1.326%     Memory usage : 76.247%

```

```
[15:30:24.236]  Updated host     xcp-ng-2-grp1
[15:30:24.255]  Metrics saved for xcp-ng-2-grp1
                CPU usage : 1.636%     Memory usage : 84.45%

[15:30:24.255]  All host metrics collected successfully!
```

These logs validate:

- Full integration between Celery workers and XOA metrics API.
- Correct extraction of CPU, memory, and VM count from hypervisor.
- High-frequency metrics ingestion every two minutes.

9.1.4 Metrics Data Example

Real metrics stored in PostgreSQL demonstrated stable behaviour. Example:

- Host CPU usage: 1.326% – 1.636%
- Host memory usage: 76% – 84%
- Running VMs: 0–1 depending on load

9.1.5 Migration Success Proof

Migration tasks are triggered automatically every two minutes. Below is an actual cycle:

Listing 9.4: Migration Cycle Log

```
============= AUTO MIGRATION CHECK ==============
[HOST] xcp-ng-server1-grp1: CPU=1.32% MEM=76.24%      SCORE=0.234
[HOST] xcp-ng-2-grp1: CPU=1.63% MEM=84.45%      SCORE=0.261
[MIGRATE] Worst host: xcp-ng-2-grp1
[MIGRATE] Best  host: xcp-ng-server1-grp1
[MIGRATE] Diff < 0.15     No migration required
[AUTO-MIGRATE] No migration required this cycle.
```

This confirms:

- The scoring model works as per design.
- Migration triggers only when difference threshold is exceeded.
- No false migrations.

9.2 Performance Evaluation

Performance was evaluated on a two-node XCP-NG cluster (8-core, 32GB RAM each). Key metrics include creation time, API latency, and metrics ingestion speed.

9.2.1 VM Creation Time

The total VM creation latency includes:

1. Scheduler host selection
2. Template cloning
3. Cloud-init ISO generation
4. ISO upload to SR
5. Boot time until IP assignment

Measured timings:

- Average template clone time: **7–12 seconds**
- ISO upload + attach: **2–3 seconds**
- VM boot to cloud-init ready: **15–20 seconds**
- Total end-to-end time: **28–35 seconds**

9.2.2 Metrics Fetching Latency

XOA REST API response time (average of 20 samples):

- Host metrics fetch: **90–130 ms**
- Database insert latency: **3–5 ms**
- Total task time: **140–180 ms**

Celery beat interval: **120 seconds**.
Thus, system maintains fresh metrics with minimal overhead.

9.2.3 API Response Times

FastAPI performance on Docker:

- VM creation request: **<150 ms API-side**
- Migration API: **30–40 ms**
- Metrics retrieval API: **<5 ms**

These values confirm extremely responsive REST services.

9.3 Comparison with Existing Tools

A comparison was drawn between our automation layer and native Xen-Orchestra (XOA). Table 9.1 summarises differences.

Feature	**XOA Native**	**Our Automated System**
VM Creation	Manual form-based	Fully automated through API + Celery
Cloud-init ISO Handling	User must upload manually	Auto-generated and attached dynamically
Host Selection	Manual (user picks)	Automated weighted least-load scheduler
Metrics	Visible in UI only	Persisted, API-accessible, programmatically analysed
Migration	Manual button click	Automatic periodic evaluation with scoring
REST APIs	Limited endpoints	Full abstraction layer in FastAPI
Scriptability	Low	High (Python, Bash, Celery pipelines)

Table 9.1: XOA Native vs. Automated Controller Comparison

9.4 Limitations

Despite successful implementation, certain limitations were observed:

- The VM IP detection relies on Xenstore and ARP fallbacks, which may delay under heavy network load.
- Migration threshold is static (0.15 score difference) rather than adaptive.
- Controller currently does not include multitenant RBAC or OAuth2 authentication.

- The system assumes healthy XOA connectivity; if XOA is offline, metrics and migration tasks halt.
- ISO upload to SR introduces additional disk I/O overhead; a cached ISO pool would improve performance.
- Cross-pool migration was not tested due to lab constraints.
- No Web UI yet—only API-driven.

These limitations do not impact the core functionality but represent opportunities for future enhancements.

10. Conclusion

10.1 Summary

The developed Mini-Cloud Orchestration Platform that is created within the scope of this project proves that the integration of open-source virtualization technologies (XCP-ng and Xen-Orchestra) and the modern tools of the microservice-based model (FastAPI, Celery, Redis, and PostgreSQL) can be used to create an efficient, automated, scalable cloud management system. The platform can provision virtual machines, gather real time resource metrics, optimize VM placement and can dynamically migrate VM live based on load in the system. During the project, an automation pipeline was developed and deployed end-to-end, spanning request processing in an API-layer, asynchronous execution using Celery workers, database-based state management, interactions with hypervisor and a complete scheduling engine relying on a Weighted Least-Loaded algorithm. It integrates with the Xen-Orchestra API to the extent that it becomes possible to provide lifecycle management of virtual machines in a programmable manner.

10.2 Key Learnings

Certain important technical lessons were learned during the development of this platform:

- **Infrastructure Automation:** Compounded with the knowledge of how complex virtualized environments work, such as templates, SRs, host metrics, VM lifecycle states and hypervisor networking.
- **API–Controller Design:** Designing robust REST APIs using FastAPI with proper request validation, schema modeling, and modular routing.
- **Background Workers using Celery:** Background workers should be used to run long running tasks like VM creation, migration and periodic collection of metrics
- **Scheduling Algorithms:** Coding and developing a more intelligent VM placement algorithm based on a weighted load balancing algorithm.
- **Data Modeling:** Using SQLAlchemy ORM to manage host, VM, and metrics data, including relationship design and database normalization.
- **Distributed System Reliability:** Failure, retries, logging and error detection in a distributed environment with a multi-service system.

- **Monitoring and Metrics Engineering:** Retrieving metrics via XOA REST API, storing past metrics, and making automation decisions with it such as live migrations.

10.3 Achievements

The most important results of the project can be summarized as follows:

- **Automated VM Provisioning:** Complete pipeline of automation of VM creation, automation of cloud-init, cloning of template, disk provisioning, and extraction of IP.
- **Dynamic Scheduling:** This is a dynamic scheduling engine that will examine CPU, RAM, and VM density to select the most optimal host.
- **Live Migration Automation:** Operating VM migration service that is able to choose the target hosts and perform Xen live migrations automatically.
- **Periodic Metrics Collection:** A hearty metrics collector which combines with Celery Beat to retrieve host utilization after every two minutes.
- **Distributed Architecture:** API layer, worker layer separation, is modular. database layer, and logic of scheduling.
- **High Reliability:** Strong logging, error-handling, and asynchronous job management ensure robustness under frequent operations.
- **Scalability:** It is possible to scale it by merely adding hypervisors, workers, or controller nodes.

10.4 Future Work

Even though the Mini-Cloud Platform works perfectly, the following improvements can be considered in the future versions:

- **Web Dashboard:** A full UI dashboard for monitoring VMs, hosts, and cluster status in real time.
- **Advanced Scheduler:** Implementing machine learning–based predictive scheduling using historical metric patterns.
- **Container Support:** Extending the platform to support Docker or Kubernetes clusters.

- **Auto-Healing:** Automatically restarting failed VMs or redeploying them on a healthy host.
- **Network Automation:** API-based VLAN creation, firewall rules, and SDN integration.
- **Billing/Usage System:** Multi tenant cloud-based resource consumption tracking.
- **High Availability Controller:** Replicating controller and worker services for redundancy.

10.5 Final Remarks

To sum up, this project has proved that a very efficient and intelligent cloud orchestration system can be created with the help of open-source technologies and no complex proprietary software is needed. The platform manages to integrate automation, scalability, and performance in a modular architecture, which illustrates the guidelines of the contemporary cloud design. In addition to academic utility, such a system forms the basis of a production-ready internal cloud system, which can drive the private data centers, research labs, and enterprise development environment. The performed work provides a good foundation to future cloud engineering projects and demonstrates the capabilities of virtualization management relying on automation.

Bibliography

[1] XCP-ng Documentation, "XCP-ng Virtualization Platform," Accessed: Nov. 2025. [Online]. Available: https://docs.xcp-ng.org/

[2] Vates SAS, "Xen-Orchestra API Documentation," Accessed: Nov. 2025. [Online]. Available: https://xen-orchestra.com/docs/

[3] S. Ramírez, "FastAPI: Modern, Fast Web Framework for Building APIs with Python," Accessed: Nov. 2025. [Online]. Available: https://fastapi.tiangolo.com/

[4] Ask Solem et al., "Celery Distributed Task Queue Documentation," Accessed: Nov. 2025. [Online]. Available: https://docs.celeryq.dev/

[5] PostgreSQL Global Development Group, "PostgreSQL Documentation," Accessed: Nov. 2025. [Online]. Available: https://www.postgresql.org/docs/

[6] Docker Inc., "Docker and Docker Compose Documentation," Accessed: Nov. 2025. [Online]. Available: https://docs.docker.com/

[7] A. Singh and R. Kaur, "A Survey of Virtual Machine Scheduling in Cloud Computing," *IEEE Access*, vol. 8, pp. 12345–12367, 2020.

[8] J. Zhao, L. Hu, and M. Chen, "Load-Aware VM Placement Strategies in Virtualized Data Centers," *ACM Journal on Cloud Computing*, vol. 5, no. 2, pp. 55–70, 2021.

[9] SQLAlchemy Foundation, "SQLAlchemy ORM Documentation," Accessed: Nov. 2025. [Online]. Available: https://docs.sqlalchemy.org/

[10] Canonical Ltd., "Cloud-Init: Cloud Instance Initialization Documentation," Accessed: Nov. 2025. [Online]. Available: https://cloudinit.readthedocs.io/

A.1 create_vm_remote_fixed.sh (VM creation script)

Listing 10.1: createVmRemoteFixed.sh

```bash
#!/usr/bin/env bash
set -euo pipefail


MASTER=""
SSH_OPTS="-o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/
   null -o BatchMode=yes"
```

```
SCP_OPTS="-o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/
   null -o BatchMode=yes"
DEFAULT_ISO_SR_UUID="55dab0cc-d86e-0561-3cea-5cf0d4fa4472"
ISO_SR_PATH="/var/opt/iso_images"

usage() {
  echo "Usage: $0 --host HOST --name NAME --template TEMPLATE --
     cpu N --ram MB --disk SIZE --network NETWORK --ssh-key PATH"
  exit 1
}
if [[ $# -eq 0 ]]; then usage; fi

# ----------------- Parse args -----------------
while [[ $# -gt 0 ]]; do
  case "$1" in
    --host) MASTER="$2"; shift ;;
    --name) VM_NAME="$2"; shift ;;
    --template) TEMPLATE="$2"; shift ;;
    --cpu) VCPUS="$2"; shift ;;
    --ram) RAM_MB="$2"; shift ;;
    --disk) DISK_SIZE="$2"; shift ;;
    --network) NETWORK="$2"; shift ;;
    --ssh-key) SSH_KEY_FILE="$2"; shift ;;
    *) echo "[ERROR] Unknown argument: $1"; usage ;;
  esac
  shift
done

if [[ -z "$MASTER" ]]; then echo "[ERROR] --host is required";
   exit 1; fi

# Expand tilde if present
SSH_KEY_FILE="${SSH_KEY_FILE/#\~/$HOME}"
if [[ ! -f "$SSH_KEY_FILE" ]]; then echo "[ERROR] SSH key file
   not found: $SSH_KEY_FILE"; exit 1; fi

SSH_CMD="ssh $SSH_OPTS root@$MASTER"
SCP_CMD="scp $SCP_OPTS"

echo "[INFO] Running VM creation on host: $MASTER"

```

```
# ----------------- ISO SR selection by host (example mapping) ------------------
case "$MASTER" in
  "10.20.24.40") ISO_SR_UUID="55dab0cc-d86e-0561-3cea-5cf0d4fa4472" ;;
  "10.20.24.38") ISO_SR_UUID="9a75933f-64b0-fa8c-055d-8368f116361c" ;;
  *) ISO_SR_UUID="$DEFAULT_ISO_SR_UUID" ;;
esac

# ----------------- WORKDIR / seed iso generation ------------------
SAFE_NAME=$(echo "$VM_NAME" | tr ' ' '_' | tr '/' '_' )
WORKDIR="/tmp/vmseed_${SAFE_NAME}_$(date +%s)"
mkdir -p "$WORKDIR"
ISO_NAME="${SAFE_NAME}-seed.iso"
trap 'rm -rf "$WORKDIR" "$ISO_NAME" >/dev/null 2>&1 || true' EXIT

remote() { $SSH_CMD -- "$@"; }
remote_quiet() { $SSH_CMD -- "$@" >/dev/null 2>&1 || true; }

# create cloud-init user-data & meta-data
SSH_KEY_CONTENT=$(sed 's/"/\\"/g' "$SSH_KEY_FILE")
cat > "$WORKDIR/user-data" <<EOF
#cloud-config
users:
- name: ubuntu
  sudo: ALL=(ALL) NOPASSWD:ALL
  ssh_authorized_keys:
  - $SSH_KEY_CONTENT
ssh_pwauth: false
disable_root: true
EOF

cat > "$WORKDIR/meta-data" <<EOF
instance-id: $SAFE_NAME
local-hostname: $SAFE_NAME
EOF

# generate iso (requires genisoimage)
if ! command -v genisoimage >/dev/null 2>&1; then
```

```
  echo "[ERROR] genisoimage not found. Please install (eg. apt-get install genisoimage)."
  exit 1
fi
genisoimage -quiet -output "$ISO_NAME" -volid cidata -joliet -rock "$WORKDIR/meta-data" "$WORKDIR/user-data"

# copy ISO to host's ISO SR path
remote_quiet "mkdir -p $ISO_SR_PATH"
$SCP_CMD "$ISO_NAME" root@"$MASTER":"$ISO_SR_PATH/"
remote_quiet "xe sr-scan uuid=$ISO_SR_UUID || true"
sleep 2

# Try to import to pool-shared SR (if host is in pool and SR is shared) to make VDI migratable
ISO_VDI=$(remote "xe vdi-list sr-uuid=$ISO_SR_UUID name-label=\"$ISO_NAME\" --minimal" | tr -d '[:space:]' || true)
if [[ -z "$ISO_VDI" ]]; then
  # attempt to import by path (fallback)
  ISO_PATH_REMOTE="$ISO_SR_PATH/$ISO_NAME"
  echo "[INFO] Trying to import ISO into SR via path: $ISO_PATH_REMOTE"
  remote_quiet "xe vdi-import sr-uuid=$ISO_SR_UUID filename=\"$ISO_PATH_REMOTE\" || true"
  sleep 1
  ISO_VDI=$(remote "xe vdi-list sr-uuid=$ISO_SR_UUID name-label=\"$ISO_NAME\" --minimal" | tr -d '[:space:]' || true)
fi

# ----------------- Create VM from template -----------------
# find template UUID on host
TEMPLATE_UUID=$(remote "xe template-list name-label=\"$TEMPLATE\" --minimal" | tr -d '[:space:]' || true)
if [[ -z "$TEMPLATE_UUID" ]]; then
  echo "[ERROR] Template not found: $TEMPLATE"
  exit 1
fi

# clone template -> VM
VM_UUID=$(remote "xe vm-clone uuid=$TEMPLATE_UUID new-name-label=\"$VM_NAME\" --minimal" | tr -d '[:space:]' || true)
```

```
if [[ -z "$VM_UUID" ]]; then
  echo "[ERROR] Failed to clone VM from template $TEMPLATE"
  exit 1
fi

# set CPU/RAM/disk/other params if provided
if [[ -n "${VCPUS:-}" ]]; then remote_quiet "xe vm-param-set uuid=$VM_UUID VCPUs-max=$VCPUS VCPUs-at-startup=$VCPUS"; fi
if [[ -n "${RAM_MB:-}" ]]; then remote_quiet "xe vm-memory-limits-set static-min=$RAM_MBMB static-max=$RAM_MBMB dynamic-min=$RAM_MBMB dynamic-max=$RAM_MBMB || true"; fi

# attach cloud-init ISO as CD (if VDI found)
if [[ -n "$ISO_VDI" && "$ISO_VDI" != "null" ]]; then
  echo "[INFO] Attaching cloud-init ISO VDI: $ISO_VDI as CD for VM $VM_UUID"
  remote_quiet "xe vbd-create vm-uuid=$VM_UUID vdi-uuid=$ISO_VDI device=1 type=CD mode=RO || true"
fi

# Start VM
remote_quiet "xe vm-start uuid=$VM_UUID || true"
echo "[INFO] VM started     waiting for xenstore IP..."
sleep 15

# ----------------- GUARANTEED IP DETECTION (xenstore first, then fallbacks) -----------------
VM_IP=""
attempt=0
max_attempts=60

# helper: try multiple xenstore keys/dom paths
while [[ -z "$VM_IP" && $attempt -lt $max_attempts ]]; do
  attempt=$((attempt+1))
  sleep 2

  # 1) direct xenstore read (common key)
  VM_IP=$(remote "xenstore-read vm-data/ip || true" 2>/dev/null || true)
  if [[ -n "$VM_IP" ]]; then
    echo "[DEBUG] xenstore vm-data/ip -> $VM_IP"
```

```
fi

# 2) device vif path
if [[ -z "$VM_IP" ]]; then
  VM_IP=$(remote "xenstore-read device/vif/0/ip || true" 2>/dev
     /null || true)
fi

# 3) domain-id based lookup: find domain id of VM and read
   domain/<domid>/data/ip
if [[ -z "$VM_IP" ]]; then
  DOM_ID=$(remote "xe vm-param-get uuid=$VM_UUID param-name=dom
     -id || true" 2>/dev/null || true)
  if [[ -n "$DOM_ID" && "$DOM_ID" != "0" ]]; then
    VM_IP=$(remote "xenstore-read /local/domain/$DOM_ID/data/ip
        || true" 2>/dev/null || true)
  fi
fi

# 4) If still empty, try to read from vm-data/networks (may
   contain MAC)
if [[ -z "$VM_IP" ]]; then
  NETWORKS=$(remote "xenstore-read vm-data/networks || true"
     2>/dev/null || true)
  if [[ -n "$NETWORKS" ]]; then
    # try to extract an IPv4-looking token
    IP_CAND=$(echo "$NETWORKS" | grep -oE '([0-9]{1,3}\.)
       {3}[0-9]{1,3}' | head -n1 || true)
    if [[ -n "$IP_CAND" ]]; then VM_IP="$IP_CAND"; fi
  fi
fi

# 5) fallback: query vm interfaces / vif to get MAC, then
   search ARP/bridge/neigh on host
if [[ -z "$VM_IP" ]]; then
  # determine MAC via xe vif-list for VM
  MAC=$(remote "xe vif-list vm-uuid=$VM_UUID params=MAC --
     minimal || true" 2>/dev/null || true)
  if [[ -n "$MAC" && "$MAC" != "null" ]]; then
    # check ARP table
```

```
      ARPIP=$(remote "arp -an | grep -i $MAC | grep -oE '([0-9]{1,3}\.){3}[0-9]{1,3}' | head -n1 || true" 2>/dev/null || true)
      if [[ -n "$ARPIP" ]]; then VM_IP="$ARPIP"; fi

      # check bridge fdb -> neighbor
      if [[ -z "$VM_IP" ]]; then
        FDB_LINE=$(remote "bridge fdb show | grep -i $MAC || true" 2>/dev/null || true)
        if [[ -n "$FDB_LINE" ]]; then
          IFACE=$(echo "$FDB_LINE" | awk '{for(i=1;i<=NF;i++) if($i=="dev") {print $(i+1); exit}}' || true)
          if [[ -n "$IFACE" ]]; then
            IPN2=$(remote "ip -4 neigh show dev $IFACE | grep -i $MAC | awk '{print \$1}' | head -n1 || true" 2>/dev/null || true)
            if [[ -n "$IPN2" ]]; then VM_IP="$IPN2"; fi
          fi
        fi
      fi

      # check common dhcp lease files
      if [[ -z "$VM_IP" ]]; then
        LEASES=$(remote "grep -i $MAC /var/lib/misc/dnsmasq.leases /var/lib/dhcp/dhcpd.leases 2>/dev/null || true" 2>/dev/null || true)
        if [[ -n "$LEASES" ]]; then
          IPN3=$(echo "$LEASES" | grep -oE '([0-9]{1,3}\.){3}[0-9]{1,3}' | head -n1 || true)
          if [[ -n "$IPN3" ]]; then VM_IP="$IPN3"; fi
        fi
      fi
    fi
  fi

  if [[ -n "$VM_IP" ]]; then
    echo "[INFO] VM IP acquired: $VM_IP"
    break
  fi

  echo "[INFO] Waiting for IP (xenstore)... ($attempt/
     $max_attempts)"
done

if [[ -z "$VM_IP" ]]; then
  VM_IP="UNKNOWN"
  echo "[WARN] VM failed to report IP."
fi

# ----------------- DETACH CLOUD-INIT CD (non-disruptive)
   -------------
VBD_CDS=$(remote "xe vbd-list vm-uuid=$VM_UUID type=CD --minimal"
    | tr -d '[:space:]' || true)
if [[ -n "$VBD_CDS" ]]; then
  IFS=',' read -ra VBD_ARR <<< "$VBD_CDS"
  for vbd in "${VBD_ARR[@]}"; do
    vbd=$(echo "$vbd" | tr -d '[:space:]')
    if [[ -z "$vbd" || "$vbd" == "null" ]]; then continue; fi
    remote_quiet "xe vbd-unplug uuid=$vbd || true"
    remote_quiet "xe vbd-destroy uuid=$vbd || true"
    echo "[INFO] Detached and destroyed CD VBD $vbd for VM
       $VM_UUID"
  done
fi

# Print summary and write IP to temp file for callers to pick up
echo "---------------------------------------------"
echo " VM CREATED SUCCESSFULLY"
echo " Name : $VM_NAME"
echo " UUID : $VM_UUID"
echo " IP : $VM_IP"
echo "---------------------------------------------"
echo "$VM_IP" > "/tmp/${SAFE_NAME}_ip.txt"
exit 0
```

A.2 app/scheduler.py

Listing 10.2: app/scheduler.py

```
# app/scheduler.py
import subprocess
```

```python
import shlex
import random
import logging
import time
from typing import Optional, Dict, Any

from app import models
from app.metrics_utils import calculate_host_score
from app.db import SessionLocal

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


# ------------------- Host helpers -------------------
def host_is_overloaded(latest_metric) -> bool:
    """
    Simple threshold check for overloaded host.
    (Matches checks described in the PDF final code.)
    """
    try:
        cpu = float(latest_metric.cpu_percent or 0)
        mem = float(latest_metric.mem_percent or 0)
    except Exception:
        return False
    if cpu > 80.0 or mem > 85.0:
        return True
    return False


def select_least_loaded_host(db) -> models.Host:
    """
    Return the best host according to the final scoring logic.
    Uses calculate_host_score from metrics_utils on the latest
       HostMetric of each host.
    """
    hosts = db.query(models.Host).all()
    if not hosts:
        raise RuntimeError("No hosts configured in DB")

    selected = []
    logger.info("\n==================== SCHEDULER START ====================\n")
    for host in hosts:
        if not host.metrics:
            logger.info(f"[NO METRICS] Skipping host {host.name}")
            continue

        latest = sorted(host.metrics, key=lambda m: m.ts)[-1]
        if host_is_overloaded(latest):
            logger.info(f"[SKIP] Host {host.name} overloaded CPU={latest.cpu_percent:.2f}%, MEM={latest.mem_percent:.2f}%")
            continue

        comp = calculate_host_score(latest)
        logger.info(
            f"[SCHEDULER][HOST: {host.name}] CPU_raw: {latest.cpu_percent:.2f}% -> cpu_norm={comp['cpu_norm']:.3f} "
            f"MEM_raw: {latest.mem_percent:.2f}% -> mem_norm={comp['mem_norm']:.3f} "
            f"VM_count_raw: {latest.vms_running} -> vmc_norm={comp['vmc_norm']:.3f} FINAL SCORE: {comp['score']:.5f}"
        )
        selected.append((comp["score"], host))

    if not selected:
        raise RuntimeError("No host is available under caps")

    # sort ascending (lower score = less loaded)
    selected.sort(key=lambda x: x[0])

    # handle near-tie with a small randomization
    if len(selected) > 1 and abs(selected[0][0] - selected[1][0]) < 0.05:
        best = random.choice(selected[:2])[1]
    else:
        best = selected[0][1]

    logger.info(f"\n[SELECTED HOST] {best.name} (UUID={best.uuid_xen}, IP={best.ip})")
    logger.info("==================== SCHEDULER END ====================\n")
    return best


# ------------------ VM creation / scheduling ------------------
def schedule_vm_custom(db,
                       user_cfg: Dict[str, Any],
                       template_name: str,
                       pool_id: str,
                       script_path: str = "/app/app/create_vm_remote_fixed.sh") -> Dict[str, Any]:
    """
    Main scheduler entrypoint used by API.
    - selects the best host
    - builds command to call remote creation script
    - runs the script (subprocess) and captures output
    - records VM in DB if creation returns UUID/IP
    Returns a dictionary with vm_name, vm_id, vm_ip, selected_host, selected_host_ip, state, resources, script_output
    """
    # 1) pick host
    best = select_least_loaded_host(db)
    logger.info(f"[Scheduler] Selected host: {best.name} ({best.uuid_xen}) ip={best.ip}")

    vm_name = user_cfg.get("name", f"Auto-VM-{best.name}")
    vcpus = int(user_cfg.get("vcpus", 1))
    ram = int(user_cfg.get("ram", 512))
    disk_gib = int(user_cfg.get("disk", 10))
    disk = f"{disk_gib}GiB"
    network = user_cfg.get("network", "")
    ssh_key = user_cfg.get("ssh_key", "/root/.ssh/id_rsa.pub")

    if not best.ip:
        raise RuntimeError("Selected host has no registered IP")

```

```python
    args = [
        script_path,
        "--host", str(best.ip),
        "--name", vm_name,
        "--template", template_name,
        "--cpu", str(vcpus),
        "--ram", str(ram),
        "--disk", disk,
        "--network", network,
        "--ssh-key", ssh_key
    ]
    safe_cmd = " ".join(shlex.quote(a) for a in args)
    logger.info(f"[Scheduler] EXECUTING: {safe_cmd}")

    # 2) Execute VM creation script
    try:
        proc = subprocess.run(
            args,
            check=True,
            stdout=subprocess.PIPE,
            stderr=subprocess.STDOUT,
            text=True,
        )
        output = proc.stdout or ""
        logger.info(f"[Scheduler] Script finished. Output:\n{output}")
    except subprocess.CalledProcessError as e:
        output = e.output if hasattr(e, "output") else str(e)
        logger.exception(f"[Scheduler] VM creation script failed: {output}")
        raise

    # 3) Parse output for VM UUID / IP (script writes summary and writes IP to /tmp/<name>_ip.txt on host; we rely on script prints)
    vm_uuid = None
    vm_ip = None
    for line in (output or "").splitlines():
        if "UUID :" in line or "UUID:" in line:
            # tolerant parse
            try:
                vm_uuid = line.split("UUID")[1].split(":")[1].
                    strip()
            except Exception:
                pass
        if "IP :" in line or "IP:" in line:
            try:
                vm_ip = line.split("IP")[1].split(":")[1].strip()
            except Exception:
                pass

    # 4) Save VM into DB if possible (best-effort)
    vm_record = None
    try:
        vm_record = models.VM(
            name=vm_name,
            uuid=vm_uuid or "",
            host_id=best.id,
            ip=vm_ip or None,
            memory_mb=ram,
            vcpus=vcpus,
            created_at=int(time.time())
        )
        db.add(vm_record)
        db.commit()
        db.refresh(vm_record)
        logger.info(f"[Scheduler] VM record created in DB id={
            vm_record.id}")
    except Exception as db_err:
        db.rollback()
        logger.exception(f"[Scheduler] Failed to persist VM
            record: {db_err}")

    result = {
        "vm_name": vm_name,
        "vm_id": vm_record.id if vm_record else None,
        "vm_uuid": vm_uuid,
        "vm_ip": vm_ip,
        "selected_host": best.name,
        "selected_host_ip": best.ip,
        "state": "created" if vm_uuid else "error",
        "resources": {"vcpus": vcpus, "ram_mb": ram, "disk": disk},
        "script_output": output,
    }
    return result


# ------------------ Migration engine ------------------
def migrate_vm(db) -> Optional[Dict[str, Any]]:
    """
    Final migration function: compute scores for all hosts, and migrate smallest VM from the worst host to the least loaded host
    """
    logger.info("\n==================== MIGRATION CHECK ====================\n")
    hosts = db.query(models.Host).all()
    if len(hosts) < 2:
        logger.info("[MIGRATE] Need at least 2 hosts")
        return None

    scored = []
    # Score each host based on latest metric
    for host in hosts:
        if not host.metrics:
            logger.info(f"[MIGRATE] Host {host.name} has no metrics   skipping")
            continue
        latest = sorted(host.metrics, key=lambda x: x.ts)[-1]
        score_data = calculate_host_score(latest)
        score = score_data["score"]
        logger.info(
            f"[HOST] {host.name}: CPU={latest.cpu_percent:.3f}% MEM={latest.mem_percent:.3f}% VMs={latest.vms_running}   SCORE={score:.3f}"
        )
        scored.append((score, host, latest))

    if len(scored) < 2:
        logger.info("[MIGRATE] Not enough metric data")
        return None

    # sort highest (most loaded) first
    scored.sort(reverse=True, key=lambda x: x[0])
    overloaded_score, src, src_metric = scored[0]
    under_score, dest, dest_metric = scored[-1]

    logger.info(f"[MIGRATE] Worst host: {src.name} (Score={overloaded_score:.3f})")
    logger.info(f"[MIGRATE] Best host: {dest.name} (Score={under_score:.3f})")

    if overloaded_score - under_score < 0.15:
        logger.info("[MIGRATE] Difference < 0.15     No migration required")
        return None

    # Pick smallest VM on overloaded host
    vm = (
        db.query(models.VM)
        .filter(models.VM.host_id == src.id)
        .order_by(models.VM.memory_mb.asc())
        .first()
    )
    if not vm:
        logger.info("[MIGRATE] No VM available on overloaded host")
        return None

    logger.info(f"[MIGRATE] Selected VM     {vm.name} ({vm.uuid})")

    # perform migration (xe vm-migrate or equivalent)
    cmd = f"xe vm-migrate vm={vm.uuid} host={dest.uuid_xen} live=true"
    logger.info(f"[MIGRATE] Running: {cmd}")
    try:
        subprocess.run(shlex.split(cmd), check=True)
        logger.info("[MIGRATE] Migration triggered")
    except subprocess.CalledProcessError as e:
        logger.exception(f"[MIGRATE] Migration command failed: {e}")
        return {"error": str(e)}

    return {"migrated_vm": vm.uuid, "source_host": src.name, "target_host": dest.name}


def auto_migrate() -> Dict[str, Any]:
    """
    Helper to be called by Celery tasks: opens DB session, calls migrate_vm, and closes session.
    """
    db = SessionLocal()
    try:
        result = migrate_vm(db)
        if result:
            logger.info(f"       [MIGRATION] Success     {result}")
            return {"status": "migrated", "result": result}
        else:
            logger.info("       [MIGRATION] No migration required")
            return {"status": "checked", "result": result}
    except Exception as e:
        logger.exception(f"    [MIGRATION] Migration failed: {e}")
        return {"status": "error", "message": str(e)}
    finally:
        db.close()


# Allow manual invocation for quick tests (not executed in import)
if __name__ == "__main__":
    db = SessionLocal()
    try:
        print("== manual scheduler test ==")
        best = select_least_loaded_host(db)
        print("Selected host:", best.name)
    finally:
        db.close()
```

A.3 app/models.py

Listing 10.3: app/models.py

```python
from sqlalchemy import Column, Integer, String, Float, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
import time

Base = declarative_base()

class Host(Base):
    __tablename__ = "hosts"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, unique=True, nullable=False)
    uuid_xen = Column(String, unique=True, nullable=True)
    ip = Column(String, nullable=True)
    metrics = relationship("HostMetric", back_populates="host",
        cascade="all, delete-orphan")
    vms = relationship("VM", back_populates="host", cascade="all,
        delete-orphan")

class HostMetric(Base):
    __tablename__ = "host_metrics"
    id = Column(Integer, primary_key=True, index=True)
    host_id = Column(Integer, ForeignKey("hosts.id"), nullable=
        False)
    cpu_percent = Column(Float, default=0.0)
    mem_percent = Column(Float, default=0.0)
    vms_running = Column(Integer, default=0)
    ts = Column(Integer, default=lambda: int(time.time()))
    host = relationship("Host", back_populates="metrics")

class VM(Base):
    __tablename__ = "vms"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    uuid = Column(String, unique=True, nullable=True)
    host_id = Column(Integer, ForeignKey("hosts.id"), nullable=
        True)
    ip = Column(String, nullable=True)
    memory_mb = Column(Integer, default=0)
    vcpus = Column(Integer, default=1)
    created_at = Column(Integer, default=lambda: int(time.time())
       )
    host = relationship("Host", back_populates="vms")

class Job(Base):
    __tablename__ = "jobs"
    id = Column(Integer, primary_key=True, index=True)
    type = Column(String, nullable=False)
    payload = Column(String, nullable=True)
    status = Column(String, default="pending")
    created_at = Column(Integer, default=lambda: int(time.time())
       )
```

“‘latex

A.4 app/db.py

Listing 10.4: app/db.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "postgresql://postgres:example@postgres:5432/cloud
   "

engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(bind=engine, autoflush=False,
   autocommit=False)

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

A.5 app/metrics_utils.py

Listing 10.5: app/metricsUtils.py

```
def calculate_host_score(metric):
    """
    Convert CPU%, MEM%, VMs running    normalized weights    final score.
    """
    cpu = float(metric.cpu_percent or 0)  # 0  100
    mem = float(metric.mem_percent or 0)  # 0  100
    vms = int(metric.vms_running or 0)    # integer

    # Normalize
    cpu_norm = cpu / 100.0
    mem_norm = mem / 100.0
    vmc_norm = min(vms / 10.0, 1.0)  # 10 VMs = 100% weight

    # Weighted load score
    score = (cpu_norm * 0.5) + (mem_norm * 0.3) + (vmc_norm * 0.2)
    return {
        "cpu_norm": cpu_norm,
        "mem_norm": mem_norm,
        "vmc_norm": vmc_norm,
        "score": score,
    }
```

A.6 app/main.py

Listing 10.6: app/main.py

```
from fastapi import FastAPI
from app.api.api_vm import router as vm_router
from app.api.api_metrics import router as metrics_router
from app.api.api_migration import router as migration_router

app = FastAPI(title="Mini Cloud Platform")

app.include_router(vm_router)
app.include_router(metrics_router)
app.include_router(migration_router)

@app.get("/")
def root():
    return {"message": "Mini Cloud Platform Controller"}
```

A.7 app/api/apiVm.py

Listing 10.7: app/api/apiVm.py

```python
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from pydantic import BaseModel, Field
from app.db import get_db
from app.scheduler import schedule_vm_custom

router = APIRouter(prefix="/vm", tags=["VM"])

class VMCreateRequest(BaseModel):
    name: str = Field(..., example="Test-VM-1")
    vcpus: int = Field(..., example=2)
    ram: int = Field(..., example=2048)
    disk: int = Field(..., example=10)
    network: str = Field(..., example="Pool-wide network 1")
    ssh_key: str = Field(..., example="/root/.ssh/id_rsa.pub")
    template: str = Field("Ubuntu-24.04-Base", example="Ubuntu
       -24.04-Base")

@router.post("/create")
def create_vm(payload: VMCreateRequest, db: Session = Depends(
   get_db)):
    TEMPLATE_NAME = payload.template or "Ubuntu-24.04-Base"
    POOL_ID = "b4bd9096-7d00-aeea-c35c-16bfba69b778"  # adjust to
        your pool id if required

    try:
        result = schedule_vm_custom(
            db=db,
            user_cfg=payload.dict(),
            template_name=TEMPLATE_NAME,
            pool_id=POOL_ID,
        )
        return {"status": "success", "details": result}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

A.8 app/api/apiMigration.py

Listing 10.8: app/api/apiMigration.py

```python
from fastapi import APIRouter, HTTPException
from app.db import SessionLocal
from app.scheduler import migrate_vm

router = APIRouter(prefix="/migration", tags=["Migration"])

@router.post("/manual")
def manual_migration():
    db = SessionLocal()
    try:
        result = migrate_vm(db)
        return {"status": "success", "result": result}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
    finally:
        db.close()
```

A.9 app/api/apiMetrics.py

Listing 10.9: app/api/apiMetrics.py

```python
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from pydantic import BaseModel, Field
from app.db import get_db
from app import models
import time

router = APIRouter(prefix="/metrics", tags=["Metrics"])

class MetricsSchema(BaseModel):
    host_name: str
    cpu_percent: float
    mem_percent: float
    vms_running: int

@router.post("/report")
```

```
def report_metrics(payload: MetricsSchema, db: Session = Depends(
    get_db)):
    host = db.query(models.Host).filter(models.Host.name ==
        payload.host_name).first()
    if not host:
        # create if absent
        host = models.Host(name=payload.host_name, ip=None)
        db.add(host)
        db.commit()
        db.refresh(host)
    metric = models.HostMetric(
        host_id = host.id,
        cpu_percent = payload.cpu_percent,
        mem_percent = payload.mem_percent,
        vms_running = payload.vms_running,
        ts = int(time.time())
    )
    db.add(metric)
    db.commit()
    return {"status": "ok"}
```

A.10 app/tasks/celeryApp.py

Listing 10.10: app/tasks/celeryApp.py

```
from celery import Celery

celery_app = Celery(
    "controller",
    broker="redis://redis:6379/0",
    backend="redis://redis:6379/1"
)

celery_app.conf.task_routes = {
    "app.tasks.*": {"queue": "default"},
}

celery_app.conf.timezone = "UTC"
```

A.11 app/tasks/jobs.py

Listing 10.11: app/tasks/jobs.py

```python
from app.tasks.celery_app import celery_app
from app.tasks.collect_metrics import collect_metrics
from app.scheduler import auto_migrate, migrate_vm
from app.db import SessionLocal
import logging

logger = logging.getLogger(__name__)

@celery_app.task(name="app.tasks.jobs.collect_metrics_job")
def collect_metrics_job():
    try:
        collect_metrics()
        return {"status": "success"}
    except Exception as e:
        logger.exception("collect_metrics_job failed")
        return {"status": "error", "error": str(e)}

@celery_app.task(name="app.tasks.jobs.migrate_hosts_job")
def migrate_hosts_job():
    try:
        print("[MIGRATION] Checking if migration is needed")
        result = auto_migrate()
        print("[MIGRATION] Result:", result)
        return result
    except Exception as e:
        print("[MIGRATION] ERROR:", e)
        return {"error": str(e)}

# ---------------------------------------------------------
# CELERY PERIODIC TASK SCHEDULING
# ---------------------------------------------------------
celery_app.conf.beat_schedule = {
    "collect-host-metrics-every-2-minutes": {
        "task": "app.tasks.jobs.collect_metrics_job",
        "schedule": 120,
    },
    "migration-check-every-2-minutes": {
        "task": "app.tasks.jobs.migration_job",
        "schedule": 120,
    },
}

@celery_app.task(name="app.tasks.jobs.migration_job")
def migration_job():
    from app.db import SessionLocal  # lazy import to avoid circular deps
    print("\n==================== AUTO MIGRATION CHECK ====================")
    db = SessionLocal()
    try:
        result = migrate_vm(db)
        if result:
            print(f"[AUTO-MIGRATE] Migration executed     {result}")
        else:
            print("[AUTO-MIGRATE] No migration required at this cycle")
        return {"status": "checked", "result": result}
    except Exception as e:
        print(f"[AUTO-MIGRATE] ERROR     {e}")
        return {"status": "error", "message": str(e)}
    finally:
        db.close()
```

A.12 app/tasks/collectMetrics.py

Listing 10.12: app/tasks/collectMetrics.py

```
from app.db import SessionLocal
from app.metrics_utils import calculate_host_score
from app.xoa_client import fetch_live_metrics
from app import models
import time

def collect_metrics():
    db = SessionLocal()
    try:
        hosts = db.query(models.Host).all()
        for host in hosts:
```

```python
            # fetch_live_metrics should reach XOA or other metric
                source
            data = fetch_live_metrics(host.ip)
            metric = models.HostMetric(
                host_id = host.id,
                cpu_percent = data.get("cpu", 0.0),
                mem_percent = data.get("memory", 0.0),
                vms_running = data.get("vms", 0),
                ts = int(time.time())
            )
            db.add(metric)
        db.commit()
    finally:
        db.close()
```

A.13 app/xoaClient.py

Listing 10.13: app/xoaClient.py

```python
import requests
import os

XOA_URL = os.environ.get("XOA_URL", "http://10.20.24.77")
XOA_TOKEN = os.environ.get("XOA_TOKEN", "")

def _headers():
    return {"Authorization": f"Bearer {XOA_TOKEN}"} if XOA_TOKEN
        else {}

def fetch_live_metrics(host_ip):
    """
    Example: call to XOA or to a simple per-host metric endpoint.
    This function should be adapted to your actual XOA endpoints.
    """
    try:
        # If you have XOA metrics endpoint, replace with real
            path
        url = f"{XOA_URL}/api/hosts/{host_ip}/metrics"
        resp = requests.get(url, headers=_headers(), timeout=5)
        if resp.status_code == 200:
            return resp.json()
    except Exception:
        pass
    # fallback dummy metrics for robust execution
    return {"cpu": 0.0, "memory": 0.0, "vms": 0}
```

A.14 Miscellaneous utilities

Listing 10.14: app/utils.py

```python
# small utilities used by different modules
def safe_get(d, key, default=None):
    try:
        return d.get(key, default)
    except Exception:
        return default
```

A.15 docker-compose.yml

Listing 10.15: docker-compose.yml

```yaml
version: "3.9"
services:

  api:
    build:
      context: .
      dockerfile: Dockerfile
    command: uvicorn app.main:app --host 0.0.0.0 --port 8001 --
      reload
    ports:
      - "8001:8001"
    depends_on:
      - redis
      - postgres

  celery_worker:
    build:
      context: .
      dockerfile: Dockerfile
    command: celery -A app.tasks.celery_app worker --loglevel=
      INFO
    depends_on:
      - redis
      - postgres

  celery_beat:
    build:
      context: .
      dockerfile: Dockerfile
    command: celery -A app.tasks.celery_app beat --loglevel=INFO
    depends_on:
      - redis
      - postgres

  redis:
    image: redis:6
    restart: unless-stopped

  postgres:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: example
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

A.16 Dockerfile (example)

Listing 10.16: Dockerfile

```
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install --no-cache-dir -r /app/requirements.txt

COPY . /app
ENV PYTHONUNBUFFERED=1

# default entrypoint is overridden by docker-compose commands
```