**Agentic AI Documentation**

# **Natural Language to OpenStack VM Creator**

This project uses **Google Gemini (LLM)** via LangChain and the **OpenStack Python SDK** to convert user natural language inputs into structured OpenStack API calls for creating virtual machines (VMs).

## **📌 Features**

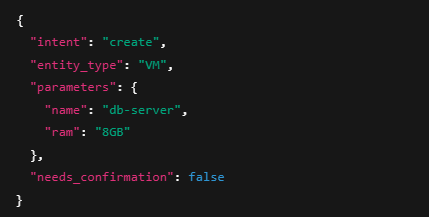
* 🧠 Natural Language Parsing with Gemini Pro 1.5
* 🔄 JSON-based intent extraction (create/delete/resize/query)
* ☁️ OpenStack API Integration for VM creation
* 🔐 Secure configuration using environment variables (.env)
* 📦 Volume-backed VM instance creation

## **🚀 Technologies Used**

* LangChain
* Google Gemini LLM
* OpenStack SDK
* Python 3.8+
* dotenv for environment variable management

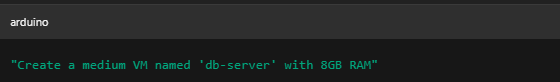
**Flow**

1. User input: *"Create a medium VM named 'db-server' with 8GB RAM*
2. Gemini parses it into:



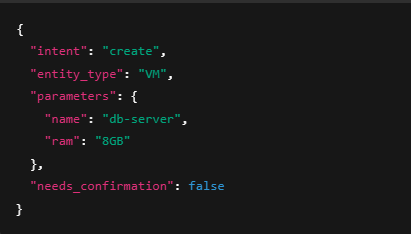
1. Mapped to OpenStack, create server API with appropriate flavor, image, and network.
2. A volume-backed VM instance is provisioned and confirmed.
3. **🔁 FLOW OF THE create FUNCTIONALITY**

### **✅ 1. User Input (Natural Language)**



### **🧠 2. Function: parse\_user\_input(user\_input)**

* **Purpose**: Converts natural language to structured data using Gemini via LangChain.
* **How**: It sends the input to Gemini with a prompt template asking for:  
  + intent (create/delete/etc.)
  + entity\_type (VM, volume, etc.)
  + parameters (e.g., name, ram)
  + needs\_confirmation (bool)



### **📤 3. Function: extract\_json(gemini\_response)**

* **Purpose**: Cleans and parses Gemini's output to extract the JSON.
* **Role**: Acts as a bridge between the language model and your logic.

### **🔄 4. Function: map\_to\_openstack\_actions(parsed\_input)**

* **Purpose**: Translates structured JSON into OpenStack-compatible API action.
* **Only triggered if intent is create and entity is VM**.
* **How**:  
  + It checks ram to determine flavor (S.4, M.8, etc.).
  + Returns an OpenStack action dictionary.



### **🔌 5. OpenStack Connection Setup**

### **🛠️ 6. VM Creation**

**Inside create\_vm():**

* **It creates a volume-backed instance using block\_device\_mapping\_v2**
* **Waits for VM to become active.**
* **Prints confirmation and details.**

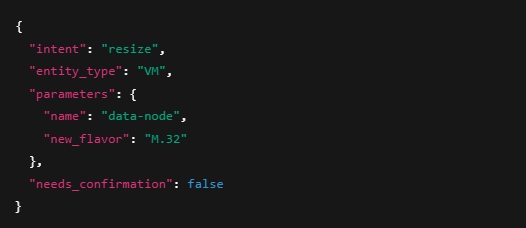
### **🔁 FLOW OF THE resize FUNCTIONALITY**

### **✅ 1. User Input (Natural Language)**

"Resize the VM named 'data-node' to M.32"

### **🧠 2. Function: parse\_user\_input(user\_input)**

Gemini (via LangChain) extracts:



### **🧼 3. Function: extract\_json(gemini\_response)**

* Parses and returns the clean JSON from the LLM output.

### **🔄 4. Intent Check**

if intent == 'resize':

### **🔌 5. Function: connect\_openstack()**

* Authenticates with OpenStack via token.
* Returns a live connection to use for further API calls.

**🔍 6. Function: find\_servers\_by\_name(conn, vm\_name)**

* **Searches for VM(s) matching the provided name.**
* **Returns a list of server objects with that name.**

### **🚦 7. Decision Making**

#### **Case A: Single VM Found**

* Resizes directly.

#### **Case B: Multiple VMs Found**

* Prompts user to choose which one to resize by index.
* Once selected, proceeds to resize.

### **⚙️ 8. Function: resize\_vm(conn, server\_id, new\_flavor\_id)**

**Steps inside:**

1. **Get the server** using ID.
2. **Call** conn.compute.resize\_server(server, new\_flavor\_id) to initiate resizing.
3. **Wait** for the VM to reach "VERIFY\_RESIZE" state.
4. **Post confirmation** with:  
   conn.compute.post(f'servers/{server\_id}/action', json={"confirmResize": None})
5. VM 'data-node' resized successfully to flavor 'M.32'.

### **3. 🔁 FLOW OF THE delete FUNCTIONALITY**

### **✅ 1. User Input (Natural Language)**

"Delete the VM named 'test-server'"

### **🧠 2. Function: parse\_user\_input(user\_input)**

{

"intent": "delete",

"entity\_type": "VM",

"parameters": {

"name": "test-server"

},

"needs\_confirmation": false

}

**🧼 3. Function: extract\_json(gemini\_response)**

* Parses and extracts the JSON object from the language model's output.

**🔄 4. Intent Handling**

if intent == 'delete':

### **🔌 5. Function: connect\_openstack()**

* Authenticates with OpenStack using token.
* Returns a valid conn object used to perform operations.

**🔍 6. Function: delete\_vm(conn, vm\_name)**

**Main logic:**

#### **a. Find VM(s)**

servers = [server for server in conn.compute.servers() if server.name == vm\_name]

* Filters all VMs with the matching name.

#### **b. Check if Found**

if not servers:

print("❌ VM not found.")

return False

#### **c. Loop and Delete**

For each matching VM:

conn.compute.delete\_server(server, ignore\_missing=False)

conn.compute.wait\_for\_delete(server)

### **📋 7. Final Result**

If all VMs deleted:  
 ✅ *VM deleted successfully*

### **4. 🌐 FLOW OF CREATE PRIVATE NETWORK ("BlueNet")**

### **✅ 1. User Input (Natural Language)**

"Create a private network called BlueNet"

### **🧠 2. Function: parse\_user\_input(user\_input)**

{

"intent": "create",

"entity\_type": "network",

"parameters": {

"name": "BlueNet",

"type": "private"

},

"needs\_confirmation": false

}

### **🧼 3. Function: extract\_json(gemini\_response)**

* Extracts clean JSON from the LLM response.

### **🔄 4. Intent Handling**

if intent == 'create' and entity\_type == 'network':

### **🔌 5. Function: connect\_openstack()**

* Connects to OpenStack using token.
* Returns the conn object to interact with the OpenStack APIs.

### **🌐 6. Function: create\_private\_network(conn, network\_name)**

**a. Check if Network Already Exists**

existing\_networks = list(conn.network.networks(name=network\_name))

Prevents duplication.

* If a network with the same name exists, it prints an error and exits.

**b. Create Network**

network = conn.network.create\_network(

name=network\_name,

is\_router\_external=False,

shared=False

)

is\_router\_external=False → marks it as **private**.

* shared=False → not visible to other tenants.

**c. Create Subnet**

subnet = conn.network.create\_subnet(

name=f"{network\_name}\_subnet",

network\_id=network.id,

ip\_version='4',

cidr='192.168.100.0/24',

gateway\_ip='192.168.100.1',

dns\_nameservers=['8.8.8.8'],

enable\_dhcp=True

)

* Creates a **/24 CIDR block** with gateway and DNS.
* DHCP is enabled for internal IP allocation.

**d. Success Message**

**✅ Private network 'BlueNet' and subnet created successfully.**

### **5. 💾 FLOW OF CREATE VOLUME ("dta" disk, 100GB)**

### **✅ 1. User Input (Natural Language)**

"Create a 100 GB volume named dta disk"

### **🧠 2. Function: parse\_user\_input(user\_input)**

{

"intent": "create",

"entity\_type": "volume",

"parameters": {

"name": "dta disk",

"size": 100

},

"needs\_confirmation": false

}

### **🧼 3. Function: extract\_json(gemini\_response)**

* Extracts valid JSON for downstream use.

### **🧠 4. Intent Check**

if entity == "volume" and intent == "create":

### **🔌 5. Function: connect\_openstack()**

* Establishes a connection with OpenStack using token-based authentication.
* Verifies with conn.authorize().
* Returns a conn object.

### **💽 6. Function: create\_volume(conn, volume\_name, size\_gb)**

**a. Log the Operation**

print(f"💾 Creating volume '{volume\_name}' of size {size\_gb} GB...")

**b. Create the Volume**

volume = conn.block\_storage.create\_volume(name=volume\_name, size=size\_gb)

**c. Wait for Completion**

conn.block\_storage.wait\_for\_status(volume, status='available', failures=['error'], interval=5, wait=300)

**d. Print Success**

print(f"✅ Volume '{volume\_name}' created successfully with ID: {volume.id}")

### **5. 💾 FLOW OF CREATE VOLUME ("dta" disk, 100GB).**

### **✅ 1. User Input (Natural Language)**

"What's my project usage?"

### **🧠 2. Function: parse\_user\_input(user\_input)**

{

"intent": "query",

"entity\_type": "project\_usage",

"parameters": {},

"needs\_confirmation": false

}

### **🧼 3. Function: extract\_json(gemini\_response)**

Extracts valid JSON used in routing logic.

### **🧠 4. Intent Check**

### if intent == "query":

### **🔌 5. Function: connect\_openstack()**

* Uses token to authenticate via OpenStack API.
* Returns conn if successful.

### **📦 6. Function: get\_project\_usage(conn)**

**a. Fetch All Servers**

servers = list(conn.compute.servers(all\_projects=False))

**b. Initialize Totals**

total\_vcpus = 0

total\_ram\_mb = 0

total\_gpus = 0

**c. Loop Over Each Server**

* Get flavor ID → fetch flavor → read vcpus, ram
* Try to extract resources:VGPU from extra\_specs

flavor = conn.compute.get\_flavor(flavor\_id)

extra\_specs = flavor.extra\_specs

gpu\_count = int(extra\_specs.get('resources:VGPU', 0))

**d. Aggregate Volume Usage**

volumes = list(conn.block\_storage.volumes(details=True))

total\_volume\_gb = sum(volume.size for volume in volumes if volume.status in ['available', 'in-use'])

**e. Print the Summary**

print(f"✅ Usage Summary:")

print(f" - Total vCPUs: {total\_vcpus}")

print(f" - Total RAM (MB): {total\_ram\_mb}")

print(f" - Total GPUs: {total\_gpus}")

print(f" - Total Volume Size (GB): {total\_volume\_gb}")

**f. Return Results**

return {

"vcpus": total\_vcpus,

"ram\_mb": total\_ram\_mb,

"gpus": total\_gpus,

"volume\_gb": total\_volume\_gb

}