IACADEMY

ENTERPRISE AGENT ENGINEERING

MÓDULO 02

Arquitecturas Multi-Tenant

Principal

iacedemy.com — 2026

MÓDULO 02

Arquitecturas Multi-Tenant

Nivel: Principal / Arquitecto

Autor: Alicia Sanz

Publicación: Julio 2026

Plataforma: iacedemy.com

Este material es parte del track Enterprise Agent Engineering de IAcademy.
Uso personal e intransferible.

> “Una plataforma de agentes sin aislamiento de tenants es una brecha de seguridad esperando a ocurrir.”

1.1 Modelos de Tenancy

El Espectro

```
Shared Everything → Shared DB → Shared Schema → Schema per Tenant → DB
per Tenant → Cluster per Tenant

(barato)
            (caro, aislado)
```

Comparativa

Modelo	Coste	Aislamiento	Complejidad	Escalado	Caso de uso
Shared Everything	Mínimo	Nulo	Baja	Vertical	Demos, free tier
Shared Database	Bajo	Bajo (RLS)	Media	Vertical	SaaS estándar
Shared Schema	Medio	Medio	Media	Horizontal	B2B mid-market
Schema per Tenant	Medio-Alto	Alto	Alta	Horizontal	Enterprise regulado
DB per Tenant	Alto	Muy alto	Alta	Horizontal	Compliance estricto
Cluster per Tenant	Máximo	Total	Muy alta	Independiente	Air-gapped, gobierno

Shared Database con RLS (Recomendado para empezar)

```
-- Supabase/PostgreSQL: Row Level Security
CREATE POLICY tenant_isolation ON agents
  USING (tenant_id = current_setting('app.tenant_id')::uuid);

-- Cada request establece el tenant
SET app.tenant_id = 'tenant-abc-123';
SELECT * FROM agents; -- Solo ve sus agentes
```

Schema per Tenant

```
# Crear schema dinámicamente al onboarding
async def provision_tenant(tenant_id: str, conn):
    schema = f"tenant_{tenant_id.replace('-', '_')}"
    await conn.execute(f"CREATE SCHEMA IF NOT EXISTS {schema}")
    await conn.execute(f"""
        CREATE TABLE {schema}.agents (
            id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
            name VARCHAR(255) NOT NULL,
            spec JSONB NOT NULL,
            status VARCHAR(20) DEFAULT 'active',
            created_at TIMESTAMPTZ DEFAULT now()
        )
    """)
    await conn.execute(f"""
        CREATE TABLE {schema}.executions (
            id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
            agent_id UUID REFERENCES {schema}.agents(id),
            input TEXT,
            output TEXT,
            tokens_used INTEGER,
            cost_eur NUMERIC(10,4),
            created_at TIMESTAMPTZ DEFAULT now()
        )
    """)
```

1.2 Aislamiento

8 Capas de Aislamiento

```
Capa 1: USUARIOS       → Auth + RBAC por tenant
Capa 2: ORGANIZACIONES → Tenant ID en cada request
Capa 3: DATOS          → RLS o schema separado
Capa 4: MODELOS        → Endpoint LLM por tenant (o compartido con
tagging)
Capa 5: HERRAMIENTAS   → Tool registry por tenant
Capa 6: SECRETOS       → Vault namespace por tenant
Capa 7: COSTES         → Budget + tracking por tenant
Capa 8: LOGS           → Tenant ID en cada traza
```

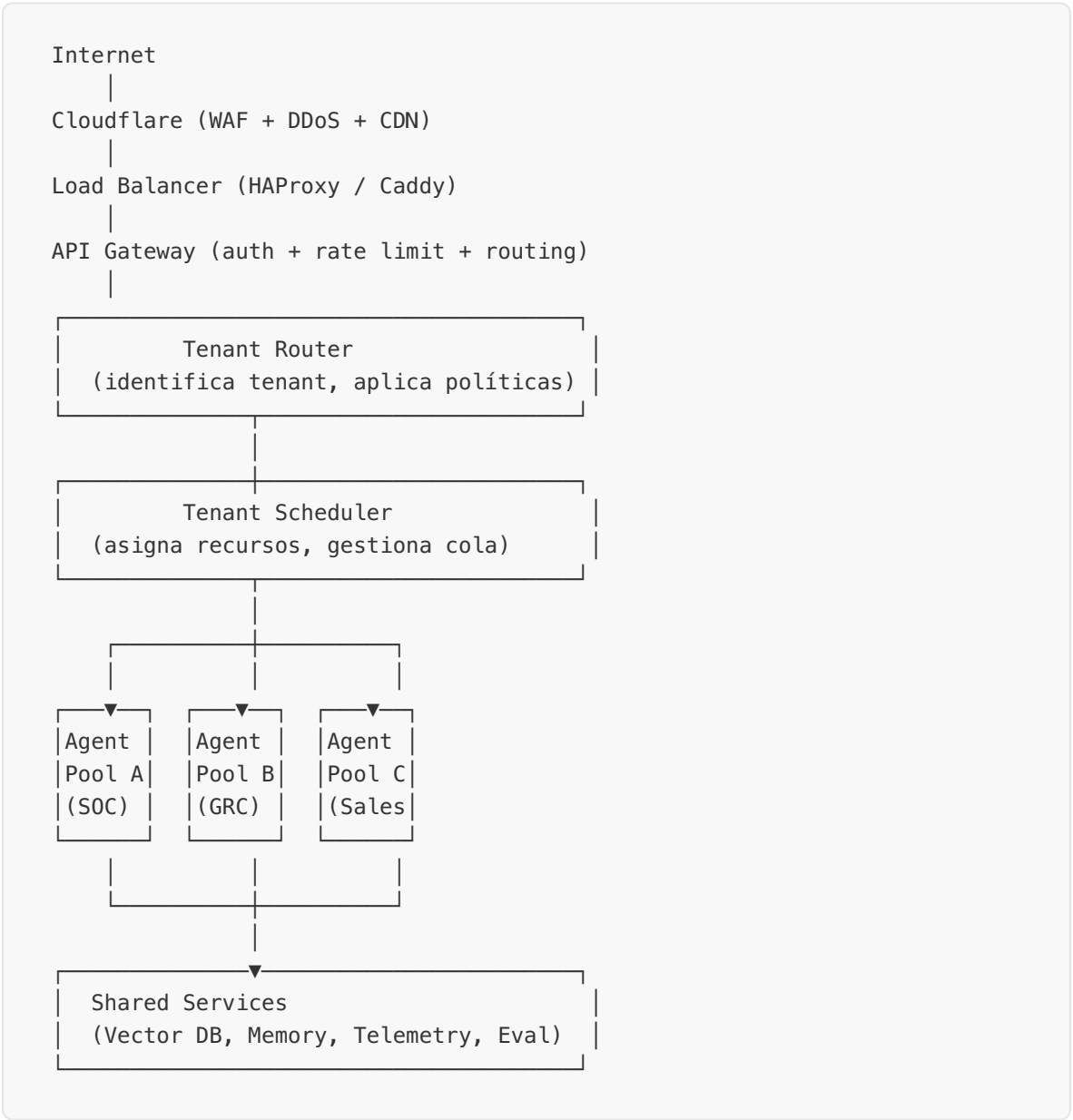
Implementación: Tenant Context

```
from contextvars import ContextVar
from dataclasses import dataclass

tenant_ctx: ContextVar[str] = ContextVar('tenant_id')

@dataclass
class TenantContext:
    tenant_id: str
    org_name: str
    plan: str  # free, pro, enterprise
    budget_eur: float
    model_endpoint: str
    vault_namespace: str

class TenantMiddleware:
    """FastAPI middleware que inyecta tenant en cada request."""

    async def __call__(self, request, call_next):
        tenant_id = request.headers.get("X-Tenant-ID")
        if not tenant_id:
            raise HTTPException(401, "Missing X-Tenant-ID")

        # Validar tenant existe
        tenant = await get_tenant(tenant_id)
        if not tenant:
            raise HTTPException(403, "Invalid tenant")

        # Inyectar en contexto
        token = tenant_ctx.set(tenant_id)
        try:
            # Configurar RLS
            await db.execute(f"SET app.tenant_id = '{tenant_id}'")
            response = await call_next(request)
            return response
        finally:
            tenant_ctx.reset(token)
```

Aislamiento de Secretos con Vault

```
import hvac

class TenantVault:
    def __init__(self, vault_url: str, token: str):
        self.client = hvac.Client(url=vault_url, token=token)

    def get_secret(self, tenant_id: str, key: str) -> str:
        path = f"tenants/{tenant_id}/{key}"
        result = self.client.secrets.kv.v2.read_secret_version(path=path)
        return result["data"]["data"]["value"]

    def set_secret(self, tenant_id: str, key: str, value: str):
        path = f"tenants/{tenant_id}/{key}"
        self.client.secrets.kv.v2.create_or_update_secret(
            path=path, secret={"value": value}
        )

    def provision_tenant(self, tenant_id: str):
        """Crear namespace de secretos para nuevo tenant."""
        # Policy que limita acceso al namespace del tenant
        policy = f"""
        path "secret/data/tenants/{tenant_id}/*" {{
            capabilities = ["create", "read", "update", "delete"]
        }}
        """
        self.client.sys.create_or_update_policy(
            name=f"tenant-{tenant_id}", policy=policy
        )
```

1.3 Arquitectura de Referencia



Tenant Router

```
from fastapi import FastAPI, Request, HTTPException
from typing import Dict

class TenantRouter:
    def __init__(self):
        self.tenants: Dict[str, dict] = {}
        self.policies: Dict[str, dict] = {}

    async def route(self, request: Request) -> dict:
        tenant_id = request.headers.get("X-Tenant-ID")

        # 1. Identificar tenant
        tenant = self.tenants.get(tenant_id)
        if not tenant:
            raise HTTPException(404, "Tenant not found")

        # 2. Verificar política
        policy = self.policies.get(tenant_id, {})
        if not self._check_policy(policy, request):
            raise HTTPException(429, "Policy violation")

        # 3. Seleccionar pool de agentes
        agent_pool = self._select_pool(tenant, request)

        # 4. Enrutar
        return {
            "tenant": tenant,
            "pool": agent_pool,
            "model_endpoint": tenant.get("model_endpoint", "default"),
            "budget_remaining": self._get_budget(tenant_id),
        }

    def _check_policy(self, policy: dict, request: Request) -> bool:
        # Rate limiting por tenant
        max_rpm = policy.get("max_requests_per_minute", 60)
        current = self._get_current_rpm(request.headers["X-Tenant-ID"])
        return current < max_rpm

    def _select_pool(self, tenant: dict, request: Request) -> str:
        # Seleccionar pool según plan del tenant
        plan = tenant.get("plan", "free")
        if plan == "enterprise":
            return "dedicated"
        return "shared"
```

1.4 Modelos de Despliegue

Decision Tree

```
¿Los datos pueden salir de la infraestructura del cliente?
  │
  ├── SÍ → ¿Cuántos tenants?
  │         ├── <10 → SaaS estándar
  │         ├── 10-100 → SaaS con isolation
  │         └── >100 → SaaS multi-region
  │
  └── NO → ¿El cliente tiene infraestructura propia?
            ├── SÍ → ¿Conectado a internet?
            │         ├── SÍ → On-Prem con updates remotos
            │         └── NO → Air-Gapped (USB updates)
            │
            └── NO → Cloud dedicado en su cuenta AWS/GCP/Azure
```

SaaS Multi-Tenant

```yaml
# docker-compose.yml — SaaS
services:
  api-gateway:
    image: caddy:latest
    ports: ["443:443"]
    volumes: ["./Caddyfile:/etc/caddy/Caddyfile"]

  tenant-router:
    image: platform/tenant-router:latest
    environment:
      - DATABASE_URL=postgresql://...
      - VAULT_URL=http://vault:8200
    deploy:
      replicas: 3

  agent-worker:
    image: platform/agent-worker:latest
    environment:
      - VLLM_ENDPOINT=http://vllm:8000
      - QDRANT_URL=http://qdrant:6333
    deploy:
      replicas: 5
      resources:
        limits: { memory: 4G, cpus: '4' }

  vllm:
    image: vllm/vllm-openai:latest
    command: --model Qwen/Qwen3.5-27B --gpu-memory-utilization 0.9
    deploy:
      resources:
        reservations:
          devices: [{ driver: nvidia, count: 1, capabilities: [gpu] }]
```

Air-Gapped

```
Particularidades:
- Sin acceso a internet
- Modelos pre-descargados en disco
- Updates via USB/medio físico
- Certificados auto-firmados
- NTP interno
- Registry de containers local (Harbor)
- Sin telemetría externa
```

1.5 Automatización de Provisioning

Terraform: Provisionar Tenant

```
# tenant.tf
resource "postgresql_schema" "tenant" {
  name     = "tenant_${var.tenant_id}"
  database = var.database_name
}

resource "vault_policy" "tenant" {
  name   = "tenant-${var.tenant_id}"
  policy = <<EOT
path "secret/data/tenants/${var.tenant_id}/*" {
  capabilities = ["create", "read", "update", "delete"]
}
EOT
}

resource "vault_token" "tenant" {
  policies = [vault_policy.tenant.name]
  ttl      = "8760h" # 1 year
}

output "tenant_config" {
  value = {
    schema      = postgresql_schema.tenant.name
    vault_token = vault_token.tenant.client_token
  }
  sensitive = true
}
```

Ansible: Deploy Agent Pool

```yaml
# deploy-agent-pool.yml
- hosts: agent_workers
  vars:
    tenant_id: "{{ tenant_id }}"
    agent_image: "platform/agent-worker:{{ version }}"

  tasks:
    - name: Pull agent image
      docker_image:
        name: "{{ agent_image }}"
        source: pull

    - name: Deploy agent container
      docker_container:
        name: "agent-{{ tenant_id }}"
        image: "{{ agent_image }}"
        env:
          TENANT_ID: "{{ tenant_id }}"
          VLLM_ENDPOINT: "http://vllm:8000"
          VAULT_TOKEN: "{{ vault_token }}"
        restart_policy: unless-stopped
        memory: "4g"
        cpus: 2
```

GitOps: ArgoCD

```yaml
# applications/tenant-abc.yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: tenant-abc
spec:
  project: tenants
  source:
    repoURL: https://github.com/platform/tenant-configs
    path: tenants/abc
    targetRevision: main
  destination:
    server: https://kubernetes.default.svc
    namespace: tenant-abc
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

1.6 Ejercicios

Ejercicio 1: Diseño de Tenancy

Tu plataforma va a servir a 50 empresas. 5 son enterprise (datos sensibles, compliance ENS). 45 son PYMEs (datos estándar). Diseña el modelo de tenancy: qué modelo para cada grupo, cómo aíslas datos, dónde pones los LLMs.

Ejercicio 2: Tenant Router

Implementa un TenantRouter completo con: rate limiting por tenant (configurable), selección de pool (shared vs dedicated), budget checking (bloquear si excede), y logging con tenant_id en cada traza.

Ejercicio 3: Provisioning Automatizado

Escribe un script Python que provisione un nuevo tenant: crear schema en PostgreSQL, namespace en Vault, configurar RLS policies, y generar token de acceso. Debe ser idempotente (ejecutar 2 veces no duplica).

Ejercicio 4: Air-Gapped Deployment

Diseña el proceso de actualización para un despliegue air-gapped: empaquetado de imágenes Docker, modelos LLM, y configuración en un medio físico. Incluye verificación de integridad (checksums) y rollback.
