IACADEMY

ENTERPRISE AGENT ENGINEERING

MÓDULO 06

FinOps para IA

Principal

iacedemy.com — 2026

MÓDULO 06

FinOps para IA

Nivel: Principal / Arquitecto

Autor: Alicia Sanz

Publicación: Julio 2026

Plataforma: iacedemy.com

> “Un buen ingeniero no usa GPT-4o para todo. Usa el modelo correcto para cada tarea.”

5.1 Anatomía del Coste

Desglose por Componente

```
COSTE POR QUERY = tokens_input × precio_input
                + tokens_output × precio_output
                + tokens_reasoning × precio_reasoning (si aplica)
                + cache_miss × precio_cache
                + GPU_seconds × precio_GPU (si self-hosted)
                + storage_bytes × precio_storage
                + bandwidth × precio_transfer
```

Precios por Provider (2026)

Provider	Modelo	Input $/1M	Output $/1M	Reasoning
Anthropic	Haiku 4.5	$0.80	$4.00	-
Anthropic	Sonnet 4.6	$3.00	$15.00	-
Anthropic	Opus 4.6	$15.00	$75.00	-
OpenAI	GPT-4o-mini	$0.15	$0.60	-
OpenAI	GPT-4o	$2.50	$10.00	-
OpenAI	o1	$15.00	$60.00	$60.00
Self-hosted	Qwen3.5-27B	$0 (GPU fijo)	$0	-
Self-hosted	Phi-4	$0 (GPU fijo)	$0	-

Coste de GPU Self-Hosted

```
# Hetzner GEX44: ~€180/mes, 48GB VRAM
# Puede servir Qwen3.5-27B con vLLM
GPU_COST_MONTHLY = 180  # EUR
QUERIES_PER_DAY = 1000
DAYS = 30

cost_per_query_selfhosted = GPU_COST_MONTHLY / (QUERIES_PER_DAY * DAYS)
# = 0.006 EUR/query

# vs API (Sonnet): ~2000 tokens/query × $3/1M input + 500 tokens × $15/1M
output
cost_per_query_api = (2000 * 3 / 1_000_000) + (500 * 15 / 1_000_000)
# = 0.0135 EUR/query

# Self-hosted es 55% más barato a 1000 queries/día
# Break-even: ~450 queries/día
```

5.2 Model Routing

Decision Tree

```
Query llega
    │
    ├── ¿Es clasificación simple? → Haiku ($0.80/1M)
    │
    ├── ¿Es generación estándar? → Sonnet ($3/1M)
    │
    ├── ¿Requiere razonamiento complejo? → Opus ($15/1M)
    │
    ├── ¿Datos sensibles? → Self-hosted Qwen ($0)
    │
    └── ¿No puede resolver? → Humano (escalación)
```

Implementación

```
from enum import Enum
from dataclasses import dataclass

class ModelTier(Enum):
    CHEAP = "cheap"
    STANDARD = "standard"
    REASONING = "reasoning"
    SOVEREIGN = "sovereign"

MODELS = {
    ModelTier.CHEAP: {"model": "claude-haiku-4-5", "cost_input": 0.80, "c
ost_output": 4.00},
    ModelTier.STANDARD: {"model": "claude-sonnet-4-6", "cost_input":
3.00, "cost_output": 15.00},
    ModelTier.REASONING: {"model": "claude-opus-4-6", "cost_input":
15.00, "cost_output": 75.00},
    ModelTier.SOVEREIGN: {"model": "qwen3.5-27b", "endpoint": "http://
vllm:8000", "cost_input": 0, "cost_output": 0},
}

class ModelRouter:
    async def select(self, task: str, sensitivity: str = "normal") ->
ModelTier:
        if sensitivity == "sensitive":
            return ModelTier.SOVEREIGN

        # Clasificar complejidad con modelo barato
        complexity = await self._classify(task)

        if complexity == "simple":
            return ModelTier.CHEAP
        elif complexity == "standard":
            return ModelTier.STANDARD
        else:
            return ModelTier.REASONING

    async def _classify(self, task: str) -> str:
        response = await llm.complete(
            model="claude-haiku-4-5",
            messages=[{"role": "user", "content": f"Classify as simple/
standard/complex: {task[:200]}"}],
            max_tokens=10,
        )
        return response.text.strip().lower()
```

5.3 Cachés

Tipos de Cache

```
import hashlib, json
import redis.asyncio as redis

class SemanticCache:
    """Cache por similitud semántica. Si una query similar ya fue
respondida, devolver cache."""

    def __init__(self, redis_client, qdrant_client, threshold=0.92):
        self.redis = redis_client
        self.qdrant = qdrant_client
        self.threshold = threshold

    async def get(self, query: str) -> str | None:
        embedding = await get_embedding(query)
        results = await self.qdrant.search("cache", embedding, limit=1)
        if results and results[0].score >= self.threshold:
            cached = await self.redis.get(f"cache:{results[0].id}")
            return cached
        return None

    async def set(self, query: str, response: str):
        embedding = await get_embedding(query)
        cache_id = hashlib.md5(query.encode()).hexdigest()
        await self.qdrant.upsert("cache", [{"id": cache_id, "vector":
embedding}])
        await self.redis.setex(f"cache:{cache_id}", 3600, response)  #
TTL 1h

class PromptCache:
    """Cache de system prompts para reducir tokens de input."""
    # Anthropic: cache_control en el system message
    # Reducción: 90% en tokens de input para prompts repetidos
    pass

class ResultCache:
    """Cache determinístico por hash de input exacto."""
    async def get(self, input_hash: str) -> str | None:
        return await self.redis.get(f"result:{input_hash}")
```

5.4 Presupuestos

Budget Manager

```
@dataclass
class Budget:
    tenant_id: str
    monthly_limit_eur: float
    daily_limit_eur: float
    per_agent_limit_eur: float
    current_monthly: float = 0
    current_daily: float = 0

class BudgetManager:
    async def check(self, tenant_id: str, estimated_cost: float) ->
tuple[bool, str]:
        budget = await self._get_budget(tenant_id)

        if budget.current_monthly + estimated_cost >
budget.monthly_limit_eur:
            return False, f"Monthly budget exceeded: {budget.current_mont
hly:.2f}/{budget.monthly_limit_eur:.2f}"

        if budget.current_daily + estimated_cost >
budget.daily_limit_eur:
            return False, f"Daily budget exceeded: {budget.current_daily:
.2f}/{budget.daily_limit_eur:.2f}"

        return True, "OK"

    async def record(self, tenant_id: str, cost: float, agent_id: str):
        await self.redis.incrbyfloat(f"budget:{tenant_id}:monthly", cost)
        await self.redis.incrbyfloat(f"budget:{tenant_id}:daily", cost)
        await self.redis.incrbyfloat(f"budget:{tenant_id}:agent:
{agent_id}", cost)

    async def alert_if_needed(self, tenant_id: str):
        budget = await self._get_budget(tenant_id)
        pct = budget.current_monthly / budget.monthly_limit_eur * 100
        if pct >= 80:
            await send_alert(tenant_id, f"Budget at {pct:.0f}%")
```

5.5 Ejercicios

Ejercicio 1: Model Router

Implementa un ModelRouter que seleccione entre 4 tiers. Incluye fallback: si el modelo principal falla, usar el siguiente tier.

Ejercicio 2: Semantic Cache

Implementa un SemanticCache con Redis + Qdrant. Mide hit rate y ahorro de costes en 100 queries de prueba.

Ejercicio 3: Budget Dashboard

Diseña un dashboard de costes por: tenant, agente, modelo, día/semana/mes. Incluye alertas al 50%, 80% y 100%.

Ejercicio 4: Optimización

Dado un escenario de 10.000 queries/día, calcula el coste mensual con: a) todo Sonnet, b) model routing, c) routing + cache. Compara.
