.UBAfiuba
FACULTAD DE INGENIERÍA

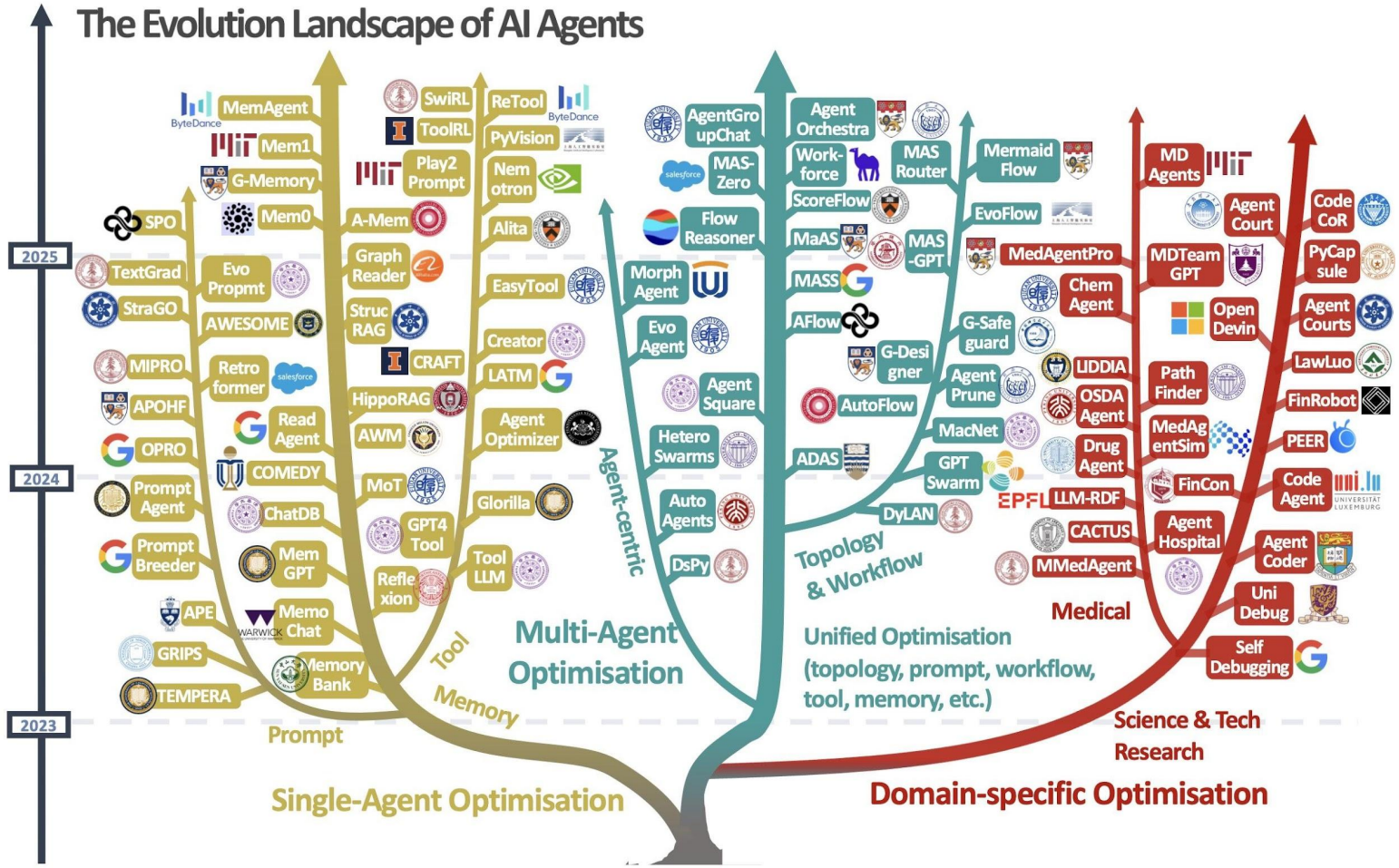
Procesamiento de Lenguaje Natural III

Docentes:
Mg. Oksana Bokhonok - FIUBA
Esp. Abraham Rodriguez - FIUBA

Programa de la materia

1. RAG avanzado y personalización de soluciones.
2. Sistemas cognitivos y agentes autónomos.
3. Seguridad. Ética y alineación de modelos con valores humanos.
4. Escalabilidad y optimización.
5. **Diseño de un sistema Agentic IA**

Taxonomía general



- Optimización de agente único: mejoras internas que perfeccionan el comportamiento de un único agente.

- Optimización multiagente: coordinación entre varios agentes para resolver tareas complejas.

- Específica de dominio: adaptación a contextos como código, legal, investigación o medicina.

GitHub - EvoAgentX/Awesome-Self-Evolving-Agents

Agentes Autónomos

Agentes que planifican, ejecutan y ajustan tareas complejas bajo restricciones de **costo, SLA (Service Level Agreement), calidad y seguridad**. **Code-first**: todo definido por políticas y flujos declarativos.

Flujo:

- Planificación jerárquica + múltiples planes.
- Ciclo *draft* ➜ *review* ➜ *execute* con autocrítica.
- Optimización de tokens, costos y latencia.
- Seguridad: sandboxing, compensaciones, watchdogs.
- Observabilidad: event-sourcing, testing, despliegues canary.
- Feedback continuo para ajuste dinámico.

Lo que buscamos:

- Automatización confiable.
- Eficiencia en recursos.
- Resiliencia y trazabilidad.
- Mejora continua sin reentrenar.

What Is an SLA (service level agreement)? | IBM

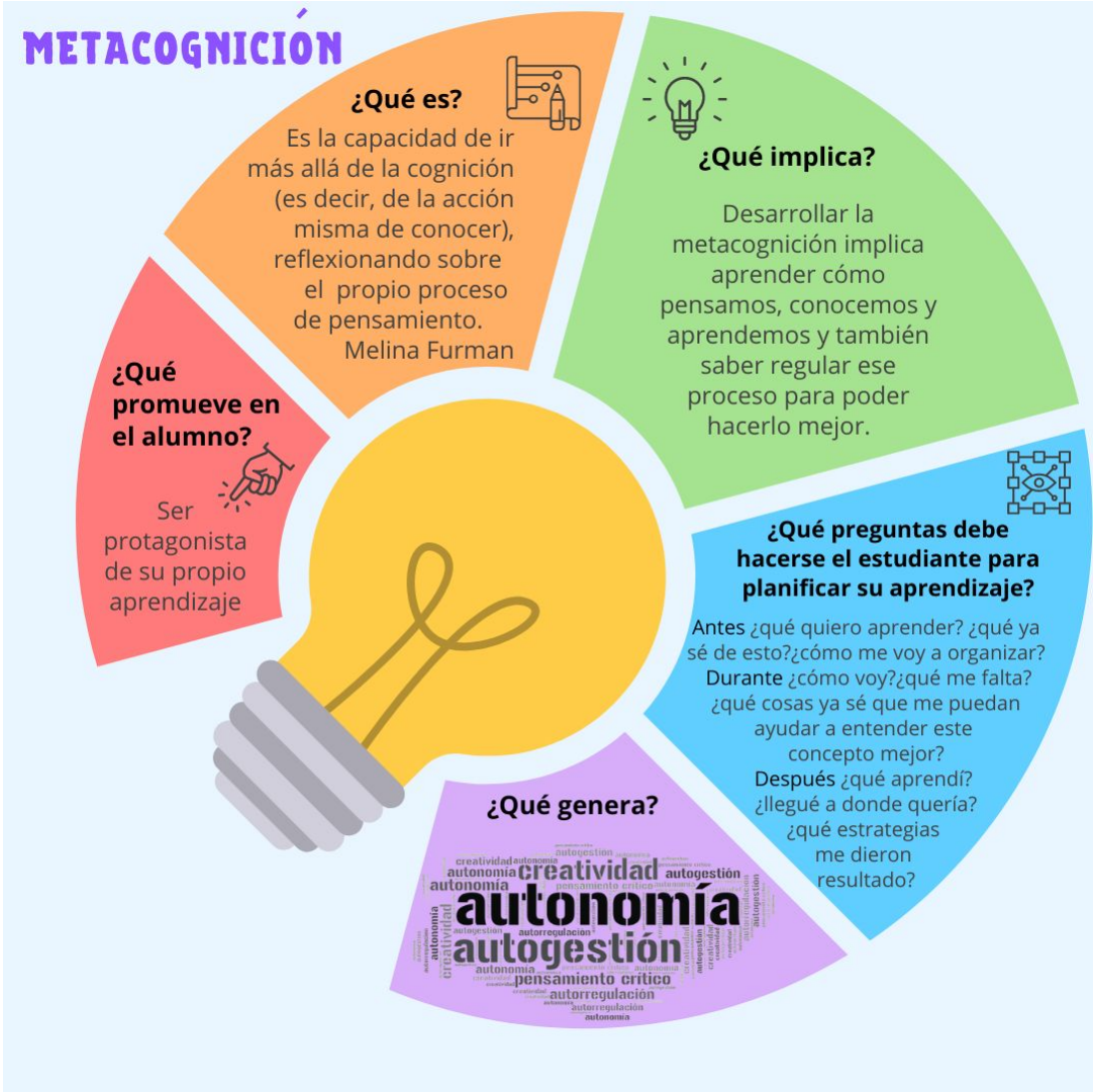
Componentes de cognición + meta-cognición

1. **Bucle draft → review → execute con autocrítica calibrada.**
2. **Gating de acciones con políticas declarativas (policy-as-code) y umbrales adaptativos.**
3. **Sandboxing de herramientas (límites de syscalls/IO/tiempo) para acciones seguras.**
4. **Statecharts (máquinas de estados con estados anidados).**
5. **Sagas y compensaciones: revertir efectos secundarios tras errores del agente.**
6. **Planificación parcial (partial-order) para paralelizar llamadas a tools sin colisiones.**
7. **Generación de múltiples planes y ranking externo con evaluadores especializados.**
8. Mantenimiento de creencias (truth-maintenance): detectar contradicciones del “mundo” del agente.
9. Gemelos digitales / simuladores para validar políticas de acción antes de producción.
10. Aprendizaje online sin gradientes: ajustar reglas/heurísticas con feedback de ejecución.
11. **Parada temprana (halting) de bucles pensamiento-acción según señales de convergencia.**
12. **Negociación entre actores automatizados (subastas/contract-net) para asignación de tareas.**
13. Recompensas jerárquicas y objetivos múltiples (precisión, tiempo, riesgo).
14. Prevención de deriva de objetivos (goal drift) con “watchdogs” semánticos.
15. **Caso integrador: adjudicación de pedidos con negociación + sagas + halting.**

Agentes con metacognición

- *La **metacognición** es la capacidad de autorregular los procesos de aprendizaje. Involucra un conjunto de operaciones intelectuales asociadas al conocimiento, control y regulación de los mecanismos cognitivos de una persona. Estos mecanismos son los que permiten recabar, evaluar y producir información, en definitiva: aprender. Según los autores más entendidos, la metacognición hace referencia a la acción y efecto de razonar sobre el propio razonamiento.*

- *Para los agentes de IA, esto significa ser capaces de evaluar y ajustar sus acciones basándose en la autoconciencia y experiencias pasadas.*



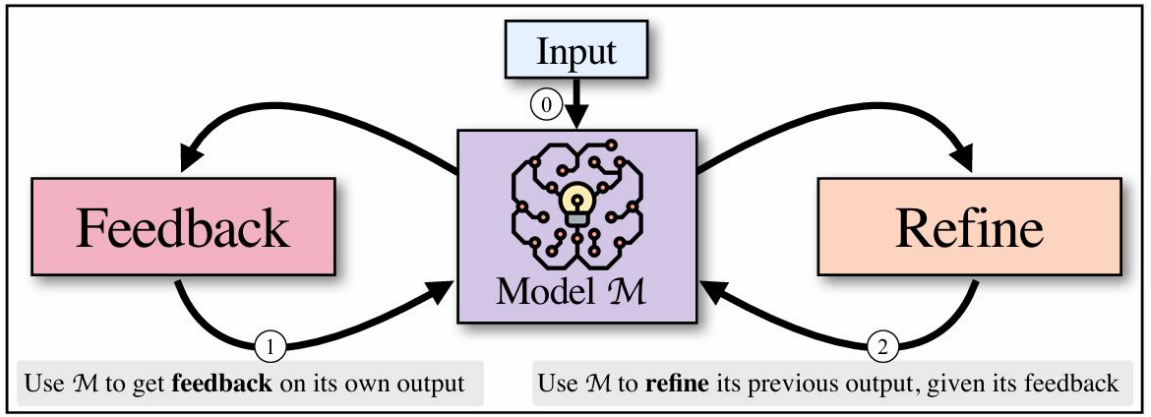
Metacognición (infografía) | Uruguay Educa

Cognición vs. Metacognición - En código / construcción de sistemas

	Cognición en código	Metacognición en código
Representaciones internas	Datos, estado del problema, modelo del mundo, memoria.	Representaciones sobre el proceso cognitivo: confianza esperada, certeza, historial de decisiones, métricas de desempeño, errores pasados.
Procesos / pipelines	Entrada → preprocesamiento → inferencia/razonamiento → salida.	"Meta-pipeline" que supervisa al pipeline cognitivo: detectar inconsistencia, baja confianza, ambigüedad → disparar reevaluación o estrategia alternativa.
Monitoreo / métricas	Logs de ejecución, tiempos, errores de cálculo.	Autoconsistencia, verificación de outputs, comparación de razonamientos, detección de contradicciones.
Control / adaptabilidad	Variaciones limitadas (algoritmo A vs. B, profundidad de búsqueda).	Decisiones meta: si confianza < umbral, cambiar modelo; retroceder si hay error; ajustar parámetros dinámicamente; aprender estrategias más eficientes.
Interfaz / feedback	Salida de resultados, API.	Explicaciones ("esto lo hago porque…"), solicitud de clarificación, autoevaluación tras feedback, registro de estrategias exitosas o fallidas.
Costo / complejidad	Menor overhead: solo procesamiento principal.	Mayor overhead: módulos extra, estimadores de confianza, retroalimentación, monitoreo adicional.

Bucle "draft → review → execute" con autocrítica calibrada.

- Generar un borrador (draft), revisarlo (review) y luego ejecutar (execute).
- Se inspira en técnicas de reflexión y autocorrección de LLMs.
- Mejora calidad reduciendo errores antes de la acción final.



(a) **Dialogue:** $x, \mathbf{y_t}$

```
User: I am interested
in playing Table
tennis.

Response: I'm sure
it's a great way to
socialize, stay active
```

(b) **FEEDBACK** $\mathbf{fb}$

```
Engaging: Provides no
information about table
tennis or how to play it.

User understanding: Lacks
understanding of user's
needs and state of mind.
```

(c) **REFINE** $\mathbf{y_{t+1}}$

```
Response (refined): That's
great to hear (...) ! It's
a fun sport requiring
quick reflexes and good
hand-eye coordination.
Have you played before, or
are you looking to learn?
```

(d) **Code optimization:** $x, \mathbf{y_t}$

```
Generate sum of 1, ..., N
def sum(n):
    res = 0
    for i in range(n+1):
        res += i
    return res
```

(e) **FEEDBACK** $\mathbf{fb}$

```
This code is slow as
it uses brute force.
A better approach is
to use the formula
... (n(n+1))/2.
```

(f) **REFINE** $\mathbf{y_{t+1}}$

```
Code (refined)

def sum_faster(n):
  return (n*(n+1))//2
```

https://arxiv.org/abs/2303.17651

https://arxiv.org/abs/2303.11366

variante sencilla del ciclo “draft → review → execute”

```python
def solve_task(prompt: str):
    response = client.chat.completions.create(
        model="gpt-4-mini",
        messages=[{"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content

result = solve_task("Resume en una frase la teoría de la relatividad.")
```

medir confianza con una función auxiliar,

reformular prompt si la confianza es baja,

comparar salidas y elegir la más fuerte,

registrar experiencia para análisis posterior,

explicar la decisión al usuario.

```python
THRESHOLD = 0.6  # confianza mínima aceptable
def estimate_confidence(output: str) -> float:
    """
    Heurística simple: mide confianza como función de la longitud,
    claridad (signos de duda) o algún proxy rápido.
    En sistemas reales, podrías usar un clasificador adicional.
    """
    if "no sé" in output.lower() or "not sure" in output.lower():
        return 0.2
    return min(1.0, 0.4 + len(output) / 100)  # muy simple

def solve_task_with_meta(prompt: str):
    # Primer intento
    response = client.chat.completions.create(
        model="gpt-4-mini",
        messages=[{"role": "user", "content": prompt}]
    )
    result = response.choices[0].message.content
    confidence = estimate_confidence(result)
    # Supervisión: ¿la confianza es suficiente?
    if confidence < THRESHOLD:
        # Estrategia alternativa: reformular prompt
        backup_prompt = f"Por favor responde de forma más clara y concisa: {prompt}"
        backup_response = client.chat.completions.create(
            model="gpt-4-mini",
            messages=[{"role": "user", "content": backup_prompt}]
        )
        alt_result = backup_response.choices[0].message.content
        alt_confidence = estimate_confidence(alt_result)

        if alt_confidence > confidence:
            result, confidence = alt_result, alt_confidence
    # Registro de experiencia (meta)
    log = {
        "prompt": prompt,
        "result": result,
        "confidence": confidence
    }
    # Explicación opcional
    explanation = f"La respuesta se seleccionó con una confianza de {confidence:.2f}."

    return result, confidence, explanation, log
result, confidence, explanation, log = solve_task_with_meta(
    "Resume en una frase la teoría de la relatividad."
)
```

variante sencilla del ciclo "draft → review → execute"


```python
THRESHOLD = 0.6  # confianza mínima aceptable
def estimate_confidence(output: str) -> float:
    """
    Heurística simple: mide confianza como función de la longitud,
    claridad (signos de duda) o algún proxy rápido.
    En sistemas reales, podrías usar un clasificador adicional.
    """
    if "no sé" in output.lower() or "not sure" in output.lower():
        return 0.2
    return min(1.0, 0.4 + len(output) / 100)  # muy simple

def solve_task_with_meta(prompt: str):
    # Primer intento
    response = client.chat.completions.create(
        model="gpt-4-mini",
        messages=[{"role": "user", "content": prompt}]
    )
    result = response.choices[0].message.content
    confidence = estimate_confidence(result)
    # Supervisión: ¿la confianza es suficiente?
    if confidence < THRESHOLD:
        # Estrategia alternativa: reformular prompt
        backup_prompt = f"Por favor responde de forma más clara y concisa: {prompt}"
        backup_response = client.chat.completions.create(
            model="gpt-4-mini",
            messages=[{"role": "user", "content": backup_prompt}]
        )
        alt_result = backup_response.choices[0].message.content
        alt_confidence = estimate_confidence(alt_result)

        if alt_confidence > confidence:
            result, confidence = alt_result, alt_confidence
    # Registro de experiencia (meta)
    log = {
        "prompt": prompt,
        "result": result,
        "confidence": confidence
    }
    # Explicación opcional
    explanation = f"La respuesta se seleccionó con una confianza de {confidence:.2f}."

    return result, confidence, explanation, log
result, confidence, explanation, log = solve_task_with_meta(
    "Resume en una frase la teoría de la relatividad."
)
```

si es apropiado **combinar enfoques**:

1. Primero un chequeo heurístico rápido (barato).
2. Si sigue dudoso → activar **self-reformulation con el LLM**.

```python
THRESHOLD = 0.6

def estimate_confidence(answer: str) -> float:
    # Supongamos que tenemos algo simple (ej: penalizar respuestas con "no sé")
    if "no sé" in answer.lower() or "not sure" in answer.lower():
        return 0.2
    return 0.7  # valor dummy para simplificar

def reformulate_with_llm(original_prompt, previous_answer):
    """El LLM reformula el prompt para obtener mejor salida."""
    response = client.chat.completions.create(
        model="gpt-4-mini",
        messages=[
            {"role": "system", "content": "Eres un experto en optimización de prompts."},
            {"role": "user", "content": f"""
            Prompt original: {original_prompt}
            Respuesta obtenida: {previous_answer}

            Identifica qué podría mejorarse y genera un nuevo prompt más claro y preciso.
            """}
        ]
    )
    return response.choices[0].message.content.strip()

def solve_task_with_self_refinement(prompt: str):
    # 1. Primer intento
    response = client.chat.completions.create(
        model="gpt-4-mini",
        messages=[{"role": "user", "content": prompt}]
    )
    result = response.choices[0].message.content
    confidence = estimate_confidence(result)

    # 2. Si la confianza es baja → pedirle al LLM que reformule
    if confidence < THRESHOLD:
        new_prompt = reformulate_with_llm(prompt, result)
        backup_response = client.chat.completions.create(
            model="gpt-4-mini",
            messages=[{"role": "user", "content": new_prompt}]
        )
        result = backup_response.choices[0].message.content
        prompt = new_prompt

    return result, prompt, confidence


# Ejemplo
result, final_prompt, confidence = solve_task_with_self_refinement(
    "Explica la teoría de la relatividad en una frase."
)
```

Despliegue ampliado del "draft → review → execute"

Supervisión y autoevaluación	Reformulación y corrección automática	Monitoreo y métricas internas
• **Estimación de confianza interna** • **Self-evaluation**: el LLM evalúa la calidad, coherencia o exactitud de su propia salida. • **Cross-checking**: comparar la respuesta con múltiples estrategias o modelos. • **Consistencia interna**: verificar que varias partes de la respuesta no se contradigan entre sí. • **Verificación con fuentes externas**: validar hechos con APIs, bases de datos o documentos confiables.	• **Prompt refinement**: el LLM genera automáticamente versiones mejoradas del prompt original. • **Parámetros adaptativos de generación**: cambiar temperature, max_tokens, top_p, etc., según confianza o dificultad. • **Uso de modelos alternativos**: fallback a otro modelo si el principal da baja confianza. • **Reescritura de respuestas**: reformular la salida para mayor claridad o precisión. • **Chain-of-Thought revisión**: revisar pasos intermedios de razonamiento antes de entregar la respuesta. • **Detección de ambigüedad**: identificar prompts o respuestas que son vagos, incompletos o ambiguos.	• **Logs de prompts y respuestas** para análisis de errores. • **Historial de decisiones**: guardar pasos previos para aprendizaje continuo. • **Rastreo de fallos recurrentes**: detectar patrones de error. • **Métricas de desempeño**: tiempo de respuesta, coherencia, cantidad de correcciones realizadas. • **Tracking de recursos**: tokens usados, costo de llamadas, uso de memoria.

Despliegue ampliado del "draft → review → execute"

Control y adaptabilidad

- **Umbrales de acción**: definir reglas para actuar si la confianza es menor a X.
- **Selección de estrategia**: decidir entre usar reasoning, retrieval-augmented generation, resumen, etc.
- **Retroalimentación adaptativa**: ajustar prompts o pipelines según resultados anteriores.
- **Aprendizaje dinámico de estrategias**: priorizar enfoques que históricamente dieron mejores resultados.
- **Rollback o replanteamiento**: retroceder pasos si se detecta inconsistencia.

Interfaz y explicación

- **Explicación de decisiones**: justificar por qué se eligió un modelo, prompt o salida.
- **Transparencia de confianza**: mostrar score de certeza al usuario.
- **Preguntas de clarificación**: pedir información adicional si el prompt es ambiguo.
- **Resumen de proceso**: generar un log legible de pasos y correcciones.

Integraciones externas / herramientas auxiliares

- **Verificación factual**: con motores de búsqueda, bases de datos, APIs de conocimiento.
- **Retrieval-Augmented Generation (RAG)**: usar documentos externos para reforzar respuestas.
- **Comparación entre modelos**: ensemble de varios LLMs y votación de outputs.
- **Simulaciones de escenarios**: probar la salida en diferentes condiciones antes de decidir.
- **Evaluadores especializados**: módulos ML que evalúan claridad, neutralidad, estilo o precisión.

Estrategias avanzadas de metacognición

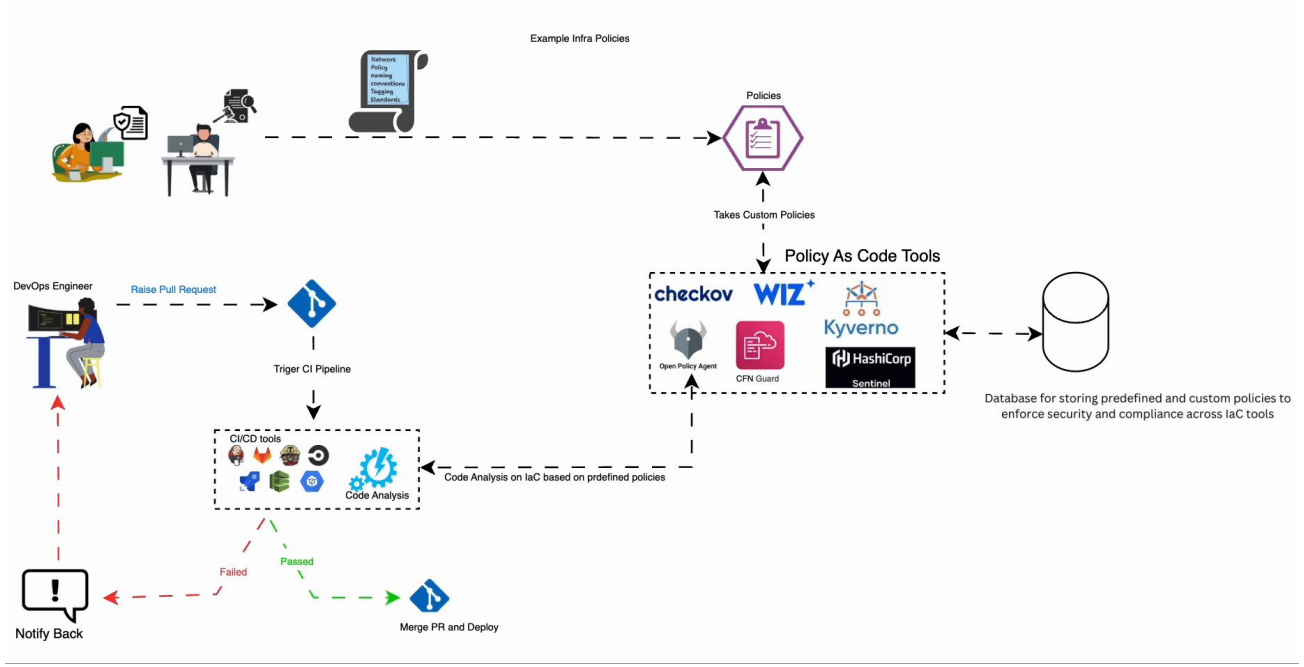
- **Auto-debugging de prompts**: detectar ambigüedades y reformular automáticamente.
- **Chain-of-thought introspective**: revisar pasos intermedios y corregir errores.
- **Learning from feedback**: usar correcciones de usuarios o métricas de desempeño para mejorar prompts y estrategias.
- **Multi-step reflection**: aplicar varias iteraciones de autoevaluación antes de entregar la respuesta final.
- **Priorizar explicabilidad vs. eficiencia**: balancear costo computacional con calidad de explicación.

Gating de acciones con políticas declarativas (policy-as-code) y umbrales adaptativos.

Policy as Code (PaC) es un enfoque donde las **políticas, reglas y estándares** de una organización se expresan como **código declarativo**. Permite automatizar decisiones, asegurando cumplimiento, seguridad y consistencia en entornos como DevOps o IaC. En esencia, PaC convierte la supervisión manual en automatización basada en reglas codificadas, integrando verificación y acción.

Ejemplos típicos:

- Validación de configuraciones de infraestructura (Infrastructure as Code) antes de desplegar.
- Enforcing reglas de seguridad y cumplimiento normativo automáticamente.
- Generación de alertas o bloqueos cuando algo viola las políticas.



[Policy As Code Tools. What is Policy as Code? | by Kudaibergenovayryska | Medium]

https://www.openpolicyagent.org/

[A practical guide to getting started with policy as code | Integration & Automation]

Policy as Code (PaC) en Agentes

Concepto general PaC	Equivalente en agentes IA/metacognitivos
Políticas codificadas declarativamente	Reglas para decisiones internas: confianza, consistencia, validación de respuestas
Evaluación de cumplimiento antes de acción	Evaluación metacognitiva de la respuesta del LLM (score de confianza, coherencia, errores)
Automatización de acción según política	Motor de Policy-as-Code decide: entregar respuesta, reformular prompt, fallback a otro modelo
Registro y aprendizaje	Logging de métricas y decisiones para ajuste de políticas y mejora continua

ai-agents-for-beginners | 12 Lessons to Get Started Building AI Agents

Learning to Deliberate: Meta-policy Collaboration for Agentic LLMs with Multi-agent Reinforcement Learning

Exploring the Potential of Metacognitive Support Agents for Human-AI Co-Creation

Sandboxing de herramientas (límites de syscalls/IO/tiempo) para acciones seguras.

Ejecutar herramientas en entornos aislados (sandbox).

Restricción de syscalls / IO / tiempo	Limitar acciones que un agente puede ejecutar (ej.: llamadas a APIs, escritura de archivos, envíos de datos)
Aislamiento de ejecución	Cada acción del agente se ejecuta en un “entorno seguro” sin afectar sistemas externos
Prevención de efectos colaterales	Evitar que decisiones o scripts generados por el LLM dañen sistemas, consuman recursos excesivos o expongan datos sensibles
Control de errores y fallos	Captura de errores y fallback automático si la acción excede límites o falla

https://gvisor.dev/

https://wasmtime.dev/

Sandboxing- algunos ejemplos simplificados

```python
import subprocess
import shlex

def run_safe_command(command, timeout=2):
    try:
        # Ejecuta en sandbox: timeout limita tiempo de ejecución
        result = subprocess.run(
            shlex.split(command),
            capture_output=True,
            text=True,
            timeout=timeout
        )
        return result.stdout
    except subprocess.TimeoutExpired:
        return "Acción abortada: timeout"
    except Exception as e:
        return f"Acción abortada: {e}"
```

```python
import multiprocessing as mp

def _runner(q, func, *args):
    try:
        q.put(func(*args))
    except Exception as e:
        q.put(f"Error: {e}")

def ejecutar_aislado(func, *args, timeout=2):
    q = mp.Queue()
    p = mp.Process(target=_runner, args=(q, func, *args))
    p.start(); p.join(timeout)
    if p.is_alive():
        p.terminate()
        return "Acción abortada: timeout"
    return q.get()
```

Statecharts (máquinas de estados con estados anidados).

Los *statecharts* extienden las máquinas de estados finitos (FSM) para **diseñar sistemas complejos** de manera clara y modular. Se usan para **estructurar el comportamiento de un agente** en estados, subestados y transiciones bien definidas.

Principios de diseño:

- **Estados jerárquicos:** un estado puede contener subestados ➔ se modela complejidad sin crecer en combinaciones planas.
- **Transiciones condicionales y jerárquicas:** simplifican la lógica al permitir reglas desde estados padre o hijo.
- **Eventos y acciones:** cada transición responde a un evento y puede disparar acciones asociadas.

Beneficios en el diseño de sistemas:

- Evitan *spaghetti code* en FSM tradicionales.
- Mejoran trazabilidad, depuración y mantenimiento.
- Permiten **modularizar la lógica del agente**, facilitando su evolución y escalabilidad.

What is a statechart? - Statecharts

https://www.sciencedirect.com/science/article/pii/0167642387900359

https://xstate.js.org/

Statecharts - Agentes IA

Estado principal	Fase general del agente (Evaluación, Acción, Logging)
Subestado	Métricas internas o estrategias dentro de la fase (confianza alta/baja, sandbox/entrega directa)
Transición	Cambio de fase según métricas, reglas declarativas o eventos (Policy as Code)
Acción en transición	Ejecución de prompt reformulado, fallback, llamada a sandbox
Jerarquía de estados	Permite modelar flujos complejos sin duplicar lógica

Statecharts- Preguntas de diseño

Evaluación / Métricas internas	- ¿Qué métricas definen confianza/riesgo y cómo se actualizan en tiempo real?- ¿Qué umbrales disparan revisión, fallback o ejecución directa?- ¿Qué reglas declarativas (*policy-as-code*) gobiernan cada acción?- ¿Cómo adaptar dinámicamente los umbrales según contexto?
Bucle draft → review → execute	- ¿Cómo implementar la revisión antes de ejecutar (self-check, reintentos)?- ¿Qué criterios disparan re-draft vs. aceptar salida?- ¿Cómo calibrar la autocrítica para balancear costo y calidad?
Acciones / Sandbox	- ¿Qué acciones están permitidas según nivel de confianza?- ¿Qué acciones requieren ejecución aislada (*sandbox*)?- ¿Qué límites de recursos (CPU, memoria, IO, tiempo) se aplican?- ¿Qué acciones son reversibles y cuáles no?- ¿Qué fallbacks deben existir?
Logging / Trazabilidad	- ¿Qué información es crítica para auditoría?- ¿Cómo guardar snapshots completos de estado (*AgentState*)?- ¿Cómo representar transiciones en un grafo histórico (*LagGraph*)?- ¿Qué datos son útiles para debugging o aprendizaje?
Statecharts / Jerarquía de estados	- ¿Qué subestados simplifican la lógica y evitan duplicación?- ¿Qué condiciones disparan transiciones jerárquicas (padre ↔ hijo)?- ¿Cómo garantizar consistencia entre estados y subestados?
Escenarios críticos	- ¿Qué hacer si una transición falla o queda bloqueada?- ¿Cómo manejar *timeouts*, errores o datos inconsistentes?- ¿Qué políticas de rollback o compensación aplicar en cada caso?

Sagas y compensaciones: revertir efectos secundarios tras errores del agente.

Una *saga* es un patrón de diseño para manejar **acciones de larga duración y con efectos externos**. Cada acción que modifica el estado del mundo tiene una **acción compensatoria** asociada, que revierte o mitiga sus efectos si algo falla más adelante. Esto permite mantener **consistencia sin transacciones bloqueantes**.

Aplicado a agentes:

- Los agentes ejecutan acciones con impacto externo (ej. publicar contenido, ejecutar comandos, modificar bases de datos).
- Si ocurre un error o la confianza del agente es baja, la saga permite **deshacer o compensar** la acción previa.
- Cada acción debe definirse junto con su acción compensatoria.

Obtenemos **reducción de riesgo de inconsistencias cuando el agente opera** de manera autónoma.

https://dl.acm.org/doi/10.1145/38713.38742

https://microservices.io/patterns/data/saga.html

Sagas y compensaciones: algunos ejemplos.

Acción agente	Efecto	Compensación
Enviar correo	Correo enviado a usuario	Enviar cancelación / retractación
Crear registro	Nuevo entry en DB	Eliminar registro
Actualizar recurso	Cambios en API externa	Revertir cambios vía API

Cada acción en un subestado puede tener una compensación asociada.

Si ocurre un fallo o el agente detecta un error vía Policy-as-Code, se dispara la transición de compensación:

- AgentState registra el fallo
- Se ejecuta la compensación
- Logging / LagGraph (grafo de estados y transiciones) registran el rollback

```python
class Agente:
    def enviar_correo(self):
        print("Correo enviado")
        # registrar en AgentState

    def compensar_correo(self):
        print("Correo retractado")
        # actualizar AgentState y LagGraph

agente = Agente()

try:
    agente.enviar_correo()
    # supongamos que algo falla después
    raise Exception("Error posterior")
except Exception as e:
    print(f"Error detectado: {e}")
    agente.compensar_correo()
```

AgentState class | Microsoft Learn

Planificación parcial (POP) para paralelizar llamadas a tools sin colisiones.

Planificación parcial es un enfoque de **IA para generar planes** donde no se fija un **orden total** de acciones desde el inicio.

- En lugar de decir “A ➜ B ➜ C” estrictamente, se define un **orden parcial**: solo se especifica lo necesario para cumplir dependencias, permitiendo que acciones independientes se ejecuten **en paralelo**.

Ejemplo conceptual:

Supongamos que un agente tiene tres acciones:

1. A: Consultar API externa
2. B: Leer base de datos interna
3. C: Generar reporte

Dependencias: C depende de A y B.

Ejemplo de herramientas:

- Google Search API ➜ necesita query antes de usarla en análisis
- Database lookup ➜ independiente de Google Search
- Email sender ➜ depende de resultados previos

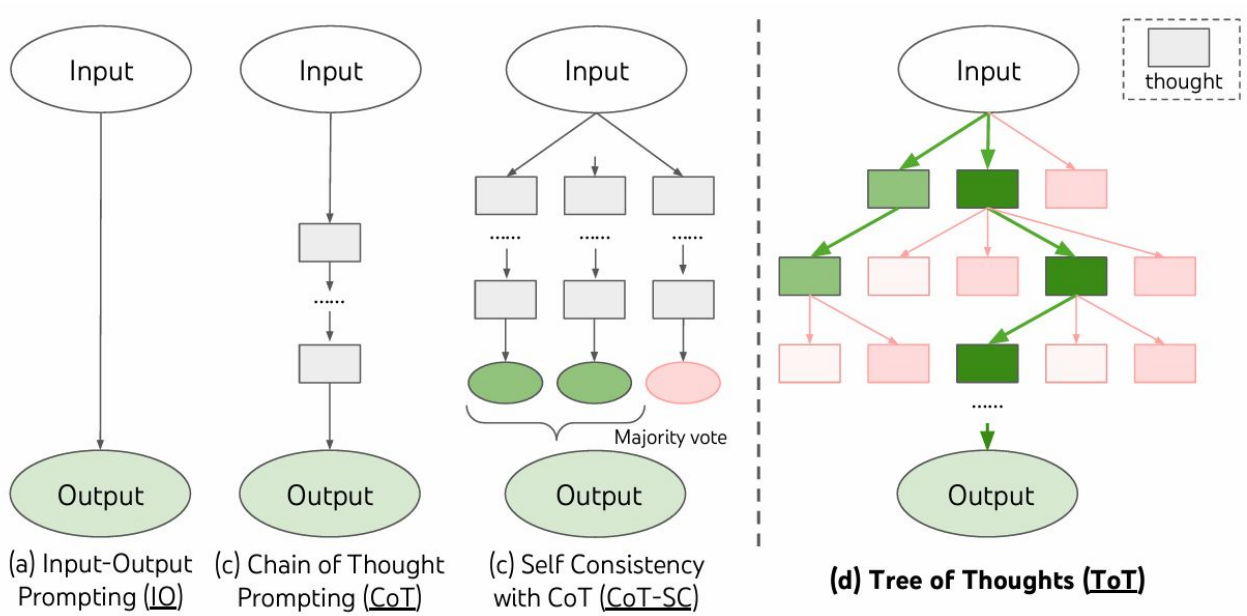
- Plan parcial: [A, B] ➜ C
- A y B pueden ejecutarse en **paralelo**, C solo después de que ambas terminen.

POP: Ejemplo conceptual en pseudocódigo

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

# Definir acciones y dependencias
actions = {
    "A": {"func": lambda: "resultado A", "depends": []},
    "B": {"func": lambda: "resultado B", "depends": []},
    "C": {"func": lambda: "resultado C", "depends": ["A", "B"]}
}

# Estado de acciones
agent_state = {k: {"status": "pending", "result": None} for k in actions}

def can_execute(action):
    return all(agent_state[d]["status"] == "done" for d in actions[action]["depends"])

executor = ThreadPoolExecutor(max_workers=2)

while any(v["status"] == "pending" for v in agent_state.values()):
    futures = {}
    for name, action in actions.items():
        if agent_state[name]["status"] == "pending" and can_execute(name):
            futures[executor.submit(action["func"])] = name

    for f in as_completed(futures):
        name = futures[f]
        agent_state[name]["result"] = f.result()
        agent_state[name]["status"] = "done"
```

Generación de múltiples planes y ranking externo con evaluadores especializados.

- El agente genera varios planes candidatos.
- Un evaluador externo (otro LLM o heurística) selecciona el mejor.
- Mejora la robustez al evitar depender de una única trayectoria.



https://arxiv.org/abs/2305.10601 (Tree of Thoughts)

ToT-Ejemplo

Generación de planes	Evaluación y ranking
El agente propone múltiples alternativas. • Plan A ➜ optimizar costo. • Plan B ➜ maximizar impacto. • Plan C ➜ balance costo/impacto. Ejemplo: un agente de marketing genera 3 campañas: • Campaña barata en redes sociales. • Evento presencial de alto impacto. • Campaña híbrida con influencers + mailing.	Evaluadores especializados asignan puntajes: • Finanzas ➜ mide costo. • Ventas ➜ estima tasa de conversión. • Legal ➜ revisa cumplimiento. Métodos de ranking: score agregado, comparaciones, comité. Ejemplo: • Plan A = barato pero bajo impacto. • Plan B = alto impacto pero caro. • Plan C = balanceado ➜ gana el ranking.

Parada temprana (halting) de bucles pensamiento-acción según señales de convergencia.

Cuando un agente entra en un ciclo *reasoning* → *acting* → *observing* → *reasoning*, puede quedarse iterando indefinidamente o gastar recursos de forma innecesaria.
La **parada temprana** es la técnica que permite **detener el ciclo antes de agotar recursos**, basándose en señales internas en lugar de un límite fijo de pasos.

Claves del enfoque:

- **Señales de convergencia (detectar redundancia):** el agente detecta cuando las nuevas iteraciones producen resultados muy similares a los anteriores (ej. embeddings casi idénticos, mismos pasos de razonamiento, poca diferencia en probabilidad del modelo)

- **Baja ganancia marginal (detectar falta de valor agregado):** identificar el punto de rendimiento decreciente, donde seguir pensando/actuando ya no agrega valor (ej. la confianza solo sube de 0.83 a 0.84, o las nuevas hipótesis no aportan información nueva)

- **Decisión dinámica (el marco general que combina las señales (incluyendo convergencia, ganancia marginal u otras) para decidir cuándo cortar):** en vez de fijar un número rígido de iteraciones (ej. 5 pasos), el agente evalúa en tiempo real cuándo frenar.

- **Otras condiciones:** Tiempo límite, presupuesto de tokens, límite de acciones....

Parada temprana (halting): ejemplo conceptual

```python
evaluators = [finance_evaluator, sales_evaluator, legal_evaluator]

# Agente con halting
class HaltingAgent:
    def __init__(self, evaluators, threshold=7.5, patience=2, max_steps=10):
        self.evaluators = evaluators
        self.threshold = threshold
        self.patience = patience
        self.max_steps = max_steps

    def generate_plan(self, step):
        plans = [
            {"name": "Plan A", "desc": "Campaña barata en redes sociales"},
            {"name": "Plan B", "desc": "Evento presencial de alto impacto"},
            {"name": "Plan C", "desc": "Campaña híbrida influencers + mailing"}
        ]
        # Para el demo: se rota entre planes
        return plans[step % len(plans)]

    def evaluate(self, plan):
        """Score real: promedio de evaluadores"""
        scores = [e(plan) for e in self.evaluators]
        return sum(scores) / len(scores)

    def run(self):
        scores = []
        reason = None
        final_plan = None

        for step in range(self.max_steps):
            plan = self.generate_plan(step)
            score = self.evaluate(plan)
            scores.append(score)
            print(f"Iteración {step+1}: {plan['name']} → score={score:.2f}")

            # Criterio 1: alcanza el umbral
            if score >= self.threshold:
                reason = f"Se alcanzó el umbral ({score:.2f} ≥ {self.threshold})"
                final_plan = plan
                break

            # Criterio 2: convergencia
            if len(scores) > self.patience:
                recent = scores[-self.patience:]
                if max(recent) - min(recent) < 0.1:  # cambio marginal mínimo
                    reason = f"Convergencia detectada (variación < 0.1 en {self.patience} pasos)"
                    final_plan = plan
                    break

        # Criterio 3: límite
        if reason is None:
            reason = "Se alcanzó el máximo de iteraciones"
            final_plan = plan

        return final_plan, scores, reason

agent = HaltingAgent(evaluators)
plan, scores, reason = agent.run()
```

Negociación entre actores automatizados (subastas/contract-net) para asignación de tareas.

Hay un conjunto de **tareas** y un conjunto de **agentes** capaces de realizarlas. * Cada agente tiene **capacidades, costos y prioridades** diferentes. * El objetivo es **asignar tareas eficientemente**, evitando conflictos y maximizando utilidad global.

Subastas (Un solo criterio)

- Un agente central (coordinador) subasta cada tarea.
- Cada agente hace **una oferta** basada en su costo, tiempo, capacidad.
- La tarea se asigna al **mejor postor** (min costo, max utilidad).

Contract-Net Protocol (CNP): Se trata de un **protocolo distribuido**.

Permite múltiples criterios de evaluación (costo, tiempo, confiabilidad, calidad), complejas.

- El agente “manager” anuncia una tarea.
- Los agentes “contractors” interesados responden con **propuestas**.
- El manager **evalúa las propuestas** y asigna la tarea.

```python
# Tarea
task = "Recolectar datos"

# Agentes con capacidades y costos
agents = {
    "Agente A": {"costo": 5, "capacidad": 10},
    "Agente B": {"costo": 3, "capacidad": 8},
    "Agente C": {"costo": 4, "capacidad": 12}
}

# Subasta simple: asignar tarea al menor costo
winner = min(agents.items(), key=lambda x: x[1]["costo"])
print(f"Tarea '{task}' asignada a {winner[0]} con costo {winner[1]['costo']}")
```

Señales y criterios de asignación

- **Costo o esfuerzo**: agente con menor costo o mayor disponibilidad.
- **Calidad**: reputación o skill level del agente.
- **Tiempo**: quién puede entregar antes.
- **Preferencias**: restricciones de prioridad o compatibilidad.

Cómo se generan esos valores dentro de un agente

- Valores preconfigurados (estáticos)

```python
class Agent:
    def __init__(self, name):
        self.name = name
        self.capacidad = 10            # número máximo de tareas que puede ejecutar
        self.costo_unitario = 5        # "precio" por realizar una tarea
        self.skills = ["Python", "SQL"] # habilidades del agente
```

- Valores calculados dinámicamente (segun estado actual):

```python
self.capacidad = max_capacity - tareas_actuales
self.costo_unitario = base_cost * factor_de_carga
if aprende_nueva_habilidad:
                    self.skills.append("SQL")
```

- Valores aprendidos o adaptativos

  En sistemas inteligentes, los agentes pueden ajustar su costo y disponibilidad según experiencia pasada:
  Si completó tareas rápido ➜ reduce costo_unitario.
  Si tiene retrasos ➜ baja su capacidad para nuevas tareas.

En un sistema distribuido: Cada agente tiene su propio estado interno (capacidad, costo, skills). Cada vez que llega una tarea, ejecuta un método tipo report_status() que devuelve esos valores. El coordinador (manager) usa esa información para decidir la asignación de tareas.

Optimización de costos

1. **Speculative decoding (base+draft) para reducir latencia/costo por token.**
2. **Early-exit / Adaptive Computation Time en decodificación.**
3. Schedules dinámicos de temperatura/top-p por paso para minimizar tokens inútiles.
4. **Bandits contextuales para elegir modelo/proveedor por request (costo-latencia-calidad).**
5. **Guardrails de presupuesto y planificación cost-aware por flujo.**
6. Truncamiento y longitudes máximas adaptativas según tipo de tarea.
7. **KV-cache policies: reemplazo, compartición y compresión (mezcla 8-bit/FP16).**
8. **Batching elástico con colas y clases de SLA (prioridades/tiempos objetivo).**
9. Decodificación asistida tipo Medusa/EAGLE-like (árboles de candidatos con verificador).
10. **Distillation operacional: enseñar endpoints críticos a modelos chicos para abaratar.**
11. Delta prompting: enviar solo diferencias respecto del contexto previo.
12. **Arbitraje de proveedores por precio/latencia en tiempo real bajo restricciones de calidad.**
13. Serving tolerante a preempciones (spot-friendly) con reanudación de estados.
14. Cómputo oportunista (ventanas off-peak y scheduling consciente de tarifas).
15. **TCO y análisis de sensibilidad: escenarios y punto de equilibrio por volumen.**

Speculative decoding (base+draft) para reducir latencia/costo por token.

Se usa un modelo rápido (draft) para proponer tokens y un modelo base para verificarlos. Reduce latencia y costo por token.

La eficiencia de speculative decoding depende de:

1. **Gap de capacidad entre draft y base:**
   - Draft debe ser lo suficientemente bueno para proponer tokens que el base aceptará frecuentemente.
   - Si draft es demasiado débil ➔ muchas correcciones ➔ menos ganancia.
   - Si draft es demasiado fuerte (casi igual que base) ➔ poca diferencia en costo.

2. **Compatibilidad en vocabulario/tokenizer:**
   - Ambos modelos deben usar la misma tokenización.
   - Si no, speculative decoding no aplica directamente (aunque se han propuesto variantes con *translation layers*).

3. **Arquitectura de inferencia:**
   - Necesita soporte en el runtime para “aceptar en batch” tokens validados, en lugar de uno por uno.
   - Frameworks como FasterTransformer, vLLM y TensorRT-LLM ya tienen implementaciones.

https://arxiv.org/abs/2302.01318

https://arxiv.org/abs/2306.04640

Early-exit / Adaptive Computation Time en decodificación.

Early-exit / ACT son técnicas que permiten detener la generación o el cálculo de un modelo antes de completarlo por completo cuando se determina que:

- La salida ya es suficientemente confiable.
- Continuar procesando aportaría un beneficio marginal mínimo.

Estas técnicas buscan finalizar la decodificación cuando la probabilidad de los tokens converge, reduciendo así pasos innecesarios de inferencia y optimizando tiempo y recursos.

```python
def adaptive_decode(prompt, max_steps=10, similarity_threshold=0.8):
    output = ""
    previous_output = ""

    for step in range(max_steps):
        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
            messages=[{"role": "user", "content": prompt + "\n\n" + output}],
            max_tokens=50
        )
        token_chunk = response.choices[0].message.content.strip()
        output += " " + token_chunk

        # Señal de confianza: similitud simple basada en palabras repetidas
        similarity = len(set(previous_output.split()) & set(output.split())) / max(1, len(output.split()))
        print(f"[Step {step+1}] Chunk: '{token_chunk[:40]}...', similarity={similarity:.2f}")

        if similarity >= similarity_threshold:
            print("→ Early-exit triggered!")
            break

        previous_output = output

    return output.strip()
```

Estrategia	Pros	Contras
Generación secuencial larga (1 llamada, 1000 tokens)	Modelo ve todo de una vez, coherencia perfecta	Latencia alta, tokens grandes de golpe, mayor riesgo de timeout
Chunk + early-exit (50 tokens x iteraciones)	Posibilidad de detenerse antes, control incremental	Reprocesa contexto anterior cada vez → overhead adicional
Draft + revisión	Draft barato + Base caro → ahorro de costo	Necesita coordinar 2 modelos y manejar prompts adicionales

Early-exit / Adaptive Computation Time en decodificación.

Cómo minimizar el sobrecosto

Si el objetivo es **contexto largo sin reprocesar todo cada vez**, podés usar estrategias como:

1. **Chunk streaming (si el modelo lo permite)**: enviar tokens de forma incremental sin volver a incluir todo el historial (actualmente limitado en API pública).

2. **Resumir el contexto acumulado**: en lugar de pasar **todo el output**, resumir o comprimir los 50–100 tokens anteriores para que el modelo tenga el "estado" esencial.

3. **Limitar early-exit a revisiones finales**: no hacer early-exit en cada mini-bloque, sino solo cuando se acerca al final del plan o respuesta.

4. **Modelos locales o LLMs con control token-level**: LLaMA, Mistral o GPT4All permiten **token streaming real** → evitas reprocesar contexto completo cada vez.

Con la API de OpenAI, **cada iteración lee todo el historial**, así que chunking excesivo puede **aumentar costos** si no se combina con técnicas de compresión de contexto o drafts baratos.

Para contextos largos (1000+ tokens), **draft + revisión** suele ser más eficiente que early-exit iterativo puro, porque el draft genera todo de una vez y el modelo caro solo valida.

Bandits contextuales para elegir modelo/proveedor por request (costo-latencia-calidad).

Selección adaptativa de proveedor/modelo según contexto (costo-latencia-calidad). Explora/explota opciones con aprendizaje online.

Cada request puede diferir en:

- **Contexto**: tipo de pregunta, complejidad, formato de entrada, prioridad, tamaño del prompt.
- **Costo**: precio por token o request.
- **Latencia**: tiempo de respuesta esperado.
- **Calidad**: exactitud, coherencia, utilidad, medida por métricas automáticas o feedback histórico.

El objetivo es **seleccionar dinámicamente la mejor opción** por request, balanceando estas métricas.

https://arxiv.org/abs/2207.02365

Guardrails de presupuesto y planificación cost-aware por flujo.

Guardrails de presupuesto: límites explícitos de **tokens, tiempo o dinero** que un agente/metacognitivo no puede superar.
Planificación cost-aware: el agente **decide acciones, selección de modelos y estrategias** considerando **costo esperado vs beneficio**, no solo calidad.

Se busca:

- Evitar gastar demasiado en inferencias de LLM.
- Priorizar rutas de procesamiento **más eficientes** (draft, early-exit, bandits contextuales).
- Mantener el flujo de decisión **seguro y predecible** desde el punto de vista de recursos.

```
[Inicio del flujo] → [Definir presupuesto total y por tarea] → [Planificación de pasos del flujo con estimación de costo] →
[Monitoreo en tiempo real del gasto] → [Ajuste dinámico: reducir tamaño, usar modelo barato, saltar pasos no críticos] →
[Reserva de contingencia para errores o revisiones] → …→ verificar gasto total ≤ presupuesto
```

KV-cache policies: reemplazo, compartición y compresión (mezcla 8-bit/FP16)

Estrategias de reemplazo y compresión en el almacenamiento KV.

Uso de 8-bit/FP16 mixto para reducir memoria.

```
[Decodificación token-by-token]
        ↓
[KV-cache] → almacenar keys/values de cada capa
        ↓
[Política de reemplazo] → truncar o descartar tokens antiguos
        ↓
[Compartición] → compartir cache entre borradores o flujos similares
        ↓
[Compresión FP16/8-bit] → reducir memoria sin perder calidad
        ↓
[Generación continua de tokens] → más eficiente en tiempo y costo
```

```python
def generate_with_kv(prompt, max_new_tokens=10, shared_kv=None):
    """
    prompt: texto inicial
    max_new_tokens: tokens a generar
    shared_kv: si queremos compartir past_key_values entre flujos
    """
    input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to(device)
    past_key_values = shared_kv  # reutiliza cache si está disponible

    generated_tokens = []

    for _ in range(max_new_tokens):
        outputs = model(input_ids=input_ids, past_key_values=past_key_values)
        logits = outputs.logits
        past_key_values = outputs.past_key_values  # KV-cache actualizada

        # seleccionar siguiente token (argmax simple)
        next_token = torch.argmax(logits[:, -1, :], dim=-1)
        generated_tokens.append(next_token.item())

        # preparar input para siguiente paso (solo el último token)
        input_ids = next_token.unsqueeze(0)

        # Ejemplo de reemplazo: truncar cache si > 5
        if past_key_values and len(past_key_values) > 5:
            past_key_values = past_key_values[-5:]

    generated_text = tokenizer.decode(generated_tokens)
    return generated_text, past_key_values
```

[KV Caching Explained: Optimizing Transformer Inference Efficiency]

Batching elástico con colas y clases de SLA (prioridades/tiempos objetivo).

Batching elástico: agrupar tareas de tamaño variable para procesarlas más eficiente, aprovechando paralelismo.

Clases de SLA: definir prioridades o tiempos objetivo para cada solicitud, asegurando que tareas críticas se atiendan primero.

Es adaptable a **cargas variables**, y facilita integración con **guardrails cost-aware y KV-cache compartida**.

[Solicitud llega] ➜ [Clasificación SLA] ➜ [Cola correspondiente]

Sigue:

- ➜ Revisar colas
- ➜ Formar batch (size <= max_batch)
- ➜ Procesar batch en paralelo
- ➜ Retornar resultados

```python
# Definición de colas SLA
# prioridad: menor número = más alta
sla_queue = PriorityQueue()

# Ejemplo de solicitudes: (prioridad, prompt)
tasks = [
    (1, "Escribe un cuento corto sobre un dragón."),
    (2, "Explica la teoría de la relatividad de forma sencilla."),
    (1, "Resume la novela Don Quijote en 3 frases."),
    (3, "Sugiere ideas para un blog de cocina vegana.")
]

# Cargar tareas en la cola
for t in tasks:
    sla_queue.put(t)

# Función de generación
def generate_text(prompt, max_tokens=20):
    input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to(device)
    output_ids = model.generate(input_ids, max_length=input_ids.shape[1]+max_tokens)
    return tokenizer.decode(output_ids[0][input_ids.shape[1]:], skip_special_tokens=True)

# Procesamiento por batches
MAX_BATCH = 2
BATCH_TIMEOUT = 1.0  # segundos

while not sla_queue.empty():
    batch = []
    start_time = time.time()
    # Formar batch elástico hasta MAX_BATCH o timeout
    while len(batch) < MAX_BATCH and not sla_queue.empty():
        priority, prompt = sla_queue.get()
        batch.append((priority, prompt))
        if time.time() - start_time > BATCH_TIMEOUT:
            break
```

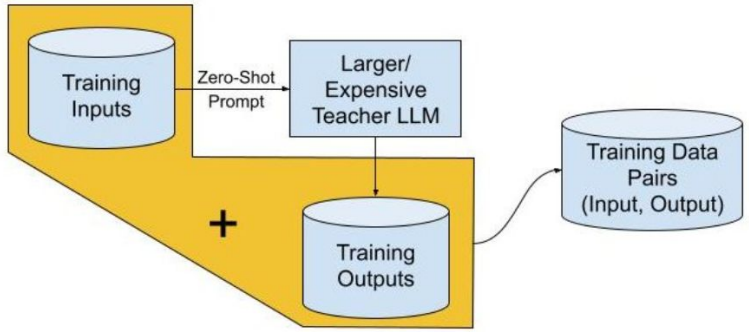
https://developer.nvidia.com/triton-inference-server

Distillation operacional: enseñar endpoints críticos a modelos chicos para abaratar.

Distillation operacional: técnica inspirada en **knowledge distillation**, donde un **modelo grande y costoso** (“teacher”) enseña a un **modelo más pequeño y económico** (“student”) para que reproduzca comportamientos importantes.

Objetivo: mantener desempeño en **endpoints críticos** (funciones o tareas clave), mientras se reduce consumo de recursos y costo por inferencia. En la práctica, la distillation operacional permite ahorrar GPU/CPU en producción porque el modelo chico atiende el **80–90% de las consultas repetitivas**, mientras el modelo grande se usa en un **10–20% de casos edge o complejos**.

1. **Identificar endpoints críticos**
   - Por ejemplo: clasificación de correos, generación de resumen ejecutivo, detección de errores graves.
   - Solo estas funciones se priorizan para entrenamiento del modelo chico.
2. **Generar dataset de distillation**
   - Se usan prompts o inputs reales que llegan a endpoints críticos.
   - Se obtiene la salida del modelo grande (“teacher”) para cada input.
3. **Entrenar modelo chico (“student”)**
   - Minimizar la diferencia entre output del teacher y student.
   - Opcional: se puede usar **soft targets** (probabilidades) para capturar incertidumbre y matices del teacher.
4. **Integración en flujo**
   - Modelo chico se usa en producción para la mayoría de tareas, especialmente las frecuentes.
   - Modelo grande se mantiene solo para casos raros o tareas no cubiertas por el student.



Arbitraje de proveedores por precio/latencia en tiempo real bajo restricciones de calidad.

Arbitraje de proveedores en tiempo real: decidir **qué proveedor/modelo usar para cada solicitud** según **precio, latencia y calidad**, optimizando costos sin sacrificar desempeño.

- Ejemplo: elegir entre GPT-4, GPT-3.5 y un modelo local económico según urgencia, presupuesto y requerimientos de precisión.

Diferencia con bandits contextuales:

- Bandits aprenden **estadísticamente qué modelo suele rendir mejor** para un contexto dado.
- Arbitraje en tiempo real **evalúa condiciones actuales** (costo, latencia, disponibilidad) y toma decisión inmediata, posiblemente combinando con bandits.

Reglas de arbitraje: Usar modelo barato si cumple calidad y tiempo objetivo. Si no, pasar a modelo más caro o rápido. Se puede combinar con SLA o prioridades de batching.

```python
import time

def call_model(provider, model, prompt):
    start = time.perf_counter()

    # Ejemplo: llamada ficticia a API
    response = provider.invoke(model=model, input=prompt)

    elapsed = time.perf_counter() - start
    log_latency(model, elapsed)
    return response

def log_latency(model, latency):
    # Se guarda en DB en lugar de hacer el print,...
    print(f"[{model}] Latencia: {latency:.2f} segundos")
```

- Comparación con ground truth (accuracy, F1, recall, precision)
- Evaluación automática sin ground truth (otro modelo grande como “evaluator”)
- Métricas semánticas (Embedding similarity, ROUGE / BLEU / METEOR, BERTScore)
- Heurísticas específicas por endpoint
- Feedback humano (cuando es crítico)

$$reward = \alpha \cdot quality - \beta \cdot price - \gamma \cdot latency$$

TCO y análisis de sensibilidad: escenarios y punto de equilibrio por volumen.

TCO (Total Cost of Ownership): costo total de operar un servicio o modelo, incluyendo:

- Costos directos: tokens, llamadas API, GPU, licencias.
- Costos indirectos: almacenamiento, mantenimiento, personal, infraestructura.
- Costos de oportunidad: retrasos por latencia, errores, retrabajo.

Análisis de sensibilidad: evaluar cómo **cambios en variables clave** (volumen de requests, costo por token, SLA) afectan el TCO o el punto de equilibrio.

Punto de equilibrio: volumen mínimo de solicitudes donde **beneficios o eficiencia justifican el costo** del modelo o inf

---

1. **Calcular TCO básico**: TCO=Costos fijos+(Costo variable por request×Volumen)
2. **Analizar sensibilidad**:
    - Variar costo por token, volumen o SLA.
    - Observar cómo cambia TCO y punto de equilibrio.
2. **Punto de equilibrio**:
    - Determinar **volumen mínimo** donde el modelo barato cubre el costo frente a modelo premium.
    - Útil para decidir **cuándo usar modelos grandes vs pequeños**.

Total Cost of Ownership (TCO) in Agentic AI | by Dr. Biraja Ghoshal | Medium

Automatización code-first

1. **DSL declarativo para describir políticas, herramientas y flujos (contratos explícitos).**
2. Verificación formal ligera de flujos (invariantes con TLA+/Alloy simplificado).
3. **Testing de contratos y property-based testing para herramientas de agentes.**
4. Fuzzing de parámetros (generadores de inputs + oráculos de invariantes).
5. **Replays deterministas con event-sourcing para depurar fallas no reproducibles.**
6. **Análisis estático/linters de prompts y de llamadas a tools para detectar anti-patrones.**
7. **Runners aislados (WASM/containers mínimos) con cuotas de CPU/Mem/IO por acción.**
8. **Despliegues prudentes: shadow/canary específicos para flujos de agente (rollback automático).**

n8n vs. Code-First: ¿cuándo elegir cada uno?

n8n / Low-Code:

- Ideal para **prototipos rápidos** y tareas simples.
- Excelente para **usuarios no técnicos**.
- Gran ecosistema de conectores listos (Gmail, Slack, Sheets, etc.).
- Útil en **automatizaciones de bajo riesgo** donde **no importa tanto la escalabilidad o el versionado**.
- La seguridad es global, no por acción. - La auditabilidad es posible, pero incómoda. - Escala, pero requiere infraestructura adicional. - Testing y despliegues controlados no son nativos.

Code-First:

- Necesario en **entornos críticos y a gran escala**.
- Seguro: cada acción corre en runners aislados con límites de recursos.
- Auditable: flujos en código versionados en Git, trazables y revisables.
- Productivo a escala: testing automático y despliegues shadow/canary que evitan errores masivos.
- Flexibilidad total para crear **tools personalizadas** sin limitaciones de plataforma.

DSL declarativo para describir políticas, herramientas y flujos (contratos explícitos).

Definir los flujos de agentes, herramientas disponibles y reglas de decisión mediante un lenguaje específico de dominio (DSL).

Evita código imperativo repetitivo y permite contratos explícitos entre componentes.

```
tool_usage:
  - tool: "GoogleSearch"
    precondition: "query_length > 3"
    cost_limit: 5
flow:
  - step: "search"
    tool: "GoogleSearch"
    output_to: "candidate_list"
  - step: "filter"
    function: "rank_candidates"
```

- Define políticas, herramientas y flujos como contratos explícitos.
- Garantiza reproducibilidad y consistencia.
- Facilita validación automática de configuraciones.

AI-DSL: Toward a General-Purpose Description Language for AI Agents | by Ben Goertzel | SingularityNET | Medium

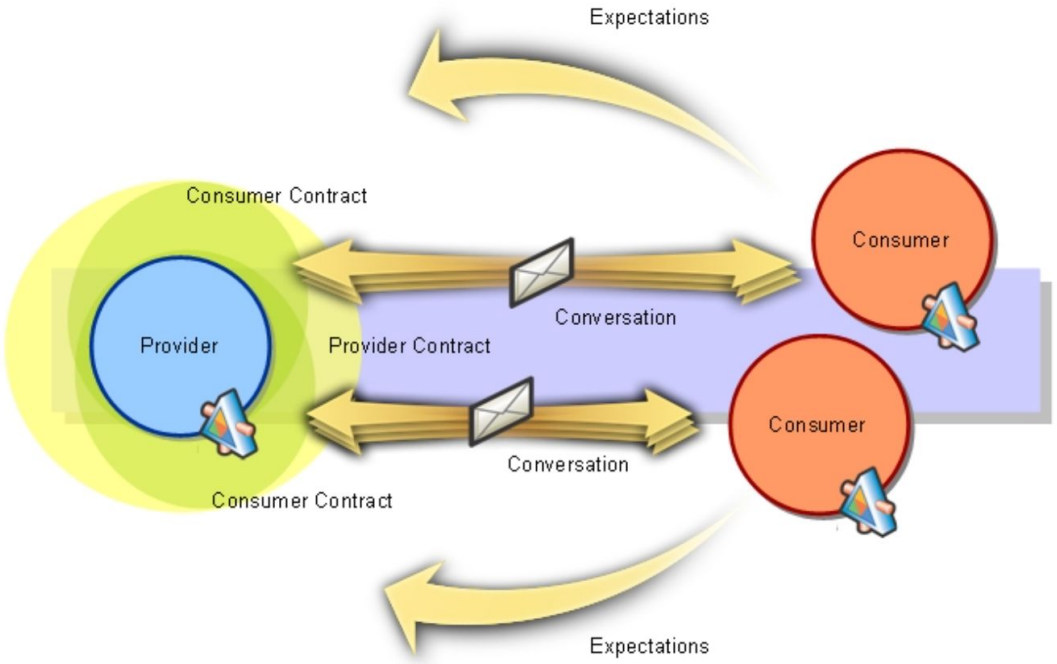
https://martinfowler.com/bliki/InfrastructureAsCode.html

Testing de contratos y property-based testing para herramientas de agentes.

- Verifica que las herramientas cumplan contratos explícitos.
- Property-based testing genera entradas aleatorias para validar invariantes.
- Detecta errores no cubiertos por tests tradicionales.

Supongamos que tenés un agente de arbitraje de modelos LLM:

- Contrato del Provider: “Para email_classification, responder en <2s y con calidad ≥0.9”.
- Property-based test: generar prompts aleatorios dentro de distintos tamaños, idiomas o formatos.
- El test verifica que el agente cumpla el contrato incluso en casos no previstos por tests tradicionales.



https://hypothesis.readthedocs.io/

https://martinfowler.com/articles/consumerDrivenContracts.html

Replays deterministas con event-sourcing para depurar fallas no reproducibles.

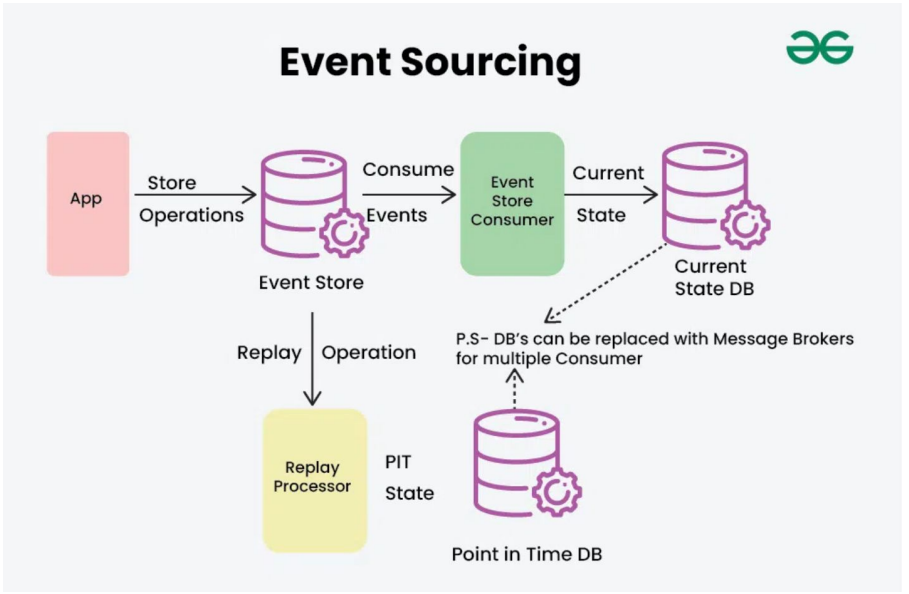
Event-Sourcing es un patrón de arquitectura donde **el estado de la aplicación no se guarda directamente**, sino que se guarda **como una secuencia de eventos** que representan todas las acciones que modifican el estado. Por ejemplo:

- Evento 1: UserCreated(id=1, name="Ana")
- Evento 2: UserEmailUpdated(id=1, email="ana@mail.com")
- Evento 3: UserDeleted(id=1)

El **estado actual** de la aplicación se puede reconstruir **reaplicando estos eventos en orden**.

Replays deterministas significa que **si vuelves a reproducir exactamente la misma secuencia de eventos, obtendrás exactamente el mismo estado**, sin variaciones. Esto permite:

- **Depuración de fallas difíciles de reproducir:** si un error ocurre en producción, puedes **reproducirlo localmente** aplicando los mismos eventos.
- **Simulación y testing:** se pueden crear escenarios completos repitiendo eventos anteriores sin afectar la producción.



https://martinfowler.com/eaaDev/EventSourcing.html

Análisis estático/linters de prompts y de llamadas a tools para detectar anti-patrones.

Revisa prompts y llamadas a tools en busca de anti-patrones:
Detecta usos ineficientes, inseguros o propensos a errores en prompts y llamadas a APIs/herramientas, por ejemplo:

- Prompts redundantes o mal formulados.
- Llamadas repetitivas a la misma herramienta sin necesidad.
- Exposición de información sensible (API keys, datos privados).

Detección temprana de problemas de consistencia o seguridad:
Permite identificar errores **antes de ejecutar el flujo completo**, como:

- Inconsistencias en formatos de datos.
- Riesgos de inyección de prompt o comandos maliciosos.

Similar a linters en desarrollo de software tradicional:
Así como un linter analiza código y sugiere mejoras o marca errores, estos sistemas revisan **flujos de LLM / agentes / pipelines** y generan alertas sobre:

- Estilo o claridad de prompts.
- Mal uso de herramientas integradas.
- Posibles fallos de lógica o seguridad.

```python
class PromptLinter:
    def __init__(self, sensitive_patterns=None, anti_patterns=None):
        # Patrón para detectar datos sensibles (ej: API keys, emails)
        self.sensitive_patterns = sensitive_patterns or [
            r"API_KEY\s*=\s*['\"][A-Za-z0-9-_]+['\"]",
            r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}"
        ]
        # Anti-patrones de prompts (ej: redundancia o malas prácticas)
        self.anti_patterns = anti_patterns or [
            r"\b(please\s+repeat)\b",  # ejemplo: redundancia
            r"\b(do not)\b.*\bignore\b"  # ejemplo de construcción confusa
        ]

    def lint_prompt(self, prompt: str):
        issues = []

        # Detectar información sensible
        for pattern in self.sensitive_patterns:
            if re.search(pattern, prompt, re.IGNORECASE):
                issues.append(f"Sensitive info detected: '{pattern}'")

        # Detectar anti-patrones
        for pattern in self.anti_patterns:
            if re.search(pattern, prompt, re.IGNORECASE):
                issues.append(f"Anti-pattern detected: '{pattern}'")

        return issues
```

Runners aislados (WASM/containers mínimos) con cuotas de CPU/Mem/IO por acción.

Runners aislados son entornos donde se ejecutan tareas o acciones de forma independiente, de modo que **un fallo o abuso en una tarea no afecta a otras** ni al sistema principal.

- **WASM (WebAssembly)**: permite ejecutar código de forma segura en un sandbox, con acceso controlado a recursos.
- **Containers mínimos**: contenedores ligeros (por ejemplo, Docker o Firecracker) que incluyen solo lo necesario para ejecutar la acción.

Cuotas de CPU/Mem/IO por acción significa que cada acción ejecutada tiene límites de recursos:

- **CPU**: porcentaje máximo de procesamiento permitido.
- **Memoria**: cantidad máxima de RAM asignada.
- **IO**: límites de lectura/escritura en disco o red.

Esto previene que:

- Una acción monopolice la CPU o memoria del host.
- Un código malicioso corrompa datos o afecte otras tareas.
- Los procesos se vuelvan inestables o provoquen denegación de servicio.

https://wasmtime.dev/

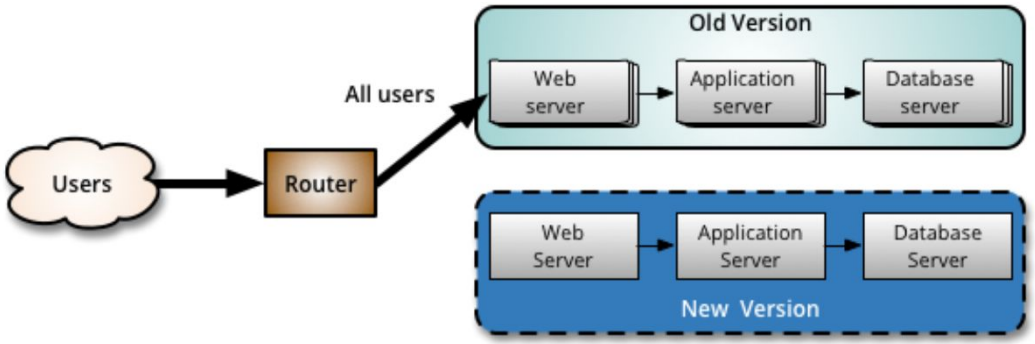
https://gvisor.dev/

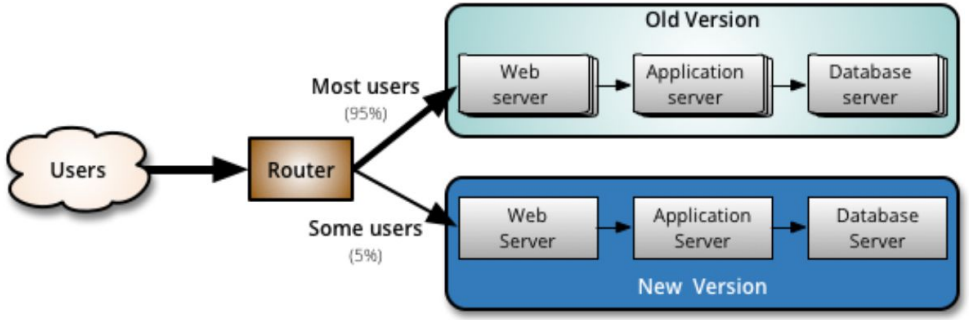
Despliegues prudentes: shadow/canary específicos para flujos de agente (rollback automático).

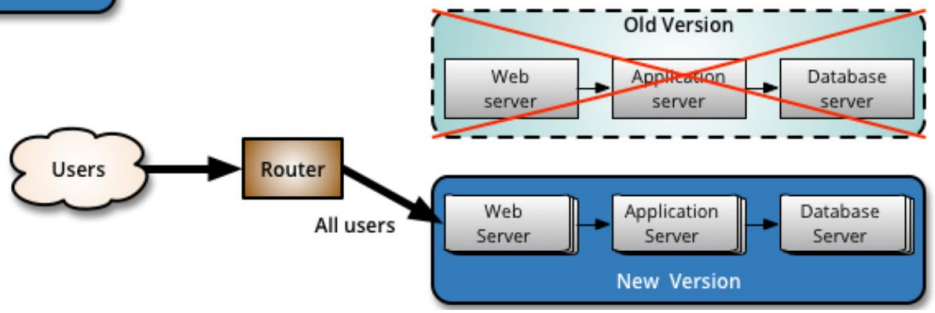
- Validación progresiva de nuevos flujos de agentes.
- Shadow: ejecución paralela sin impacto en usuarios.
- Canary: despliegue controlado a un subconjunto.
- Rollback automático en caso de fallos

https://martinfowler.com/bliki/CanaryRelease.html

https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar







[Despliegue canario]

1. Definición del flujo y políticas
   └─ DSL declarativo ➜ flujos, herramientas, contratos explícitos, restricciones de costo

2. Verificación previa
   ├─ Linters / análisis estático ➜ detecta anti-patrones, errores de prompts, uso indebido de herramientas
   ├─ Testing de contratos / property-based ➜ valida invariantes, inputs/outputs
   └─ Optional: verificación formal ligera (TLA+/Alloy simplificado)

3. Preparación y optimización de ejecución
   ├─ KV-cache policies ➜ compartición, reemplazo, compresión de embeddings
   ├─ Arbitraje de proveedores ➜ selección por costo, latencia y calidad mínima
   ├─ Guardrails de presupuesto y planificación cost-aware ➜ control de gastos por request/flujo
   ├─ Distillation operacional ➜ uso de modelos chicos en endpoints críticos
   └─ Bandits contextuales / TCO analysis ➜ ajustar elección de modelo según desempeño histórico y volumen

4. Formado de batches y ejecución
   ├─ Batching elástico ➜ grupos de requests según SLA y timeout
   ├─ Early-exit / adaptive computation time ➜ parar cálculos cuando convergen resultados
   ├─ Speculative decoding ➜ reducir latencia/costo en generación de tokens
   └─ Runners aislados (WASM/containers) ➜ cuotas de CPU/Mem/IO por acción

5. Ejecución del flujo
   ├─ Subastas / contract-net ➜ negociación de tareas entre agentes
   ├─ Planificación jerárquica (HTN) ➜ pasos complejos y dependencias
   ├─ Draft ➜ review ➜ execute con autocrítica calibrada ➜ mejora iterativa
   ├─ Sagas y compensaciones ➜ revertir efectos secundarios si falla un paso
   └─ Statecharts ➜ manejar estados y transiciones complejas

6. Monitoreo y registro
   ├─ Event-sourcing / replays deterministas ➜ debug y reproducciones
   ├─ Registro de métricas: costo, latencia, calidad, consumo de recursos
   ├─ Mantenimiento de creencias ➜ detectar contradicciones del “mundo” del agente
   └─ Gemelos digitales / simuladores ➜ validar políticas antes de producción

7. Despliegue y feedback
   ├─ Shadow / Canary deployments ➜ probar flujos nuevos de manera segura
   ├─ Rollback automático si métricas fallan
   └─ Aprendizaje online sin gradientes ➜ ajustar reglas/heurísticas con feedback de ejecución

Gracias por su atención y dedicación.

Recuerden que los grandes retos traen grandes aprendizajes.

¡Nos vemos en la próxima clase!