Sistemes Multiagent amb MCP

Eloi Puertas i Prats
Universitat de Barcelona
Grau en Enginyeria Informàtica
201-05-2026

Índex de continguts

1. Fonaments

- Evolució dels paradigmes
- Agents clàssics vs Agents LLM
- Arquitectures multi-agent

2. Cas d'Estudi: Fish Auction

- Escenari real
- Arquitectura distribuïda
- Pattern Fan-out/Fan-in

3. Model Context Protocol (MCP)

- Protocol estandarditzat
- Arquitectura MCP
- 4 Gaps arquitectònics i solucions

4. Skills dels Agents

- Què són els Skills
- Diferència entre Skills i Tools
- Arquitectura amb Skills

5. Implementació Pràctica

- Frameworks (LangChain, LangGraph, CrewAI)
- Local LLM vs API
- Demos funcionals

6. Observabilitat i Producció

- Logging estructurat
- Debugging d'agents
- Best practices

1. Fonaments

- Evolució dels paradigmes
- Agents clàssics vs Agents LLM
- Arquitectures multi-agent

Evolució dels paradigmes

```
Client / Server
        ↓
Web Services
        ↓
 Microservices
        ↓
LLM Agents + MCP
```

La idea clau:

> els agents NO substitueixen els sistemes distribuïts: **els compliquen**.

Ara afegim:

> *reasoning probabilístic*

Per tant:

> El software **interpreta**. No únicament executa.
> Ara els missatges entre agents són **semàntics**.

Arquitectura clàssica vs Arquitecutura moderna

Client / servidor

```
Client
   |
 request
   |
Server
   |
 response
   |
Client
```

El flux és:

- explícit
- programat
- determinista

Sistema multiagent LLM

```
Coordinator
     |
 +---+---+----------+
 |   |   |          |
Buyer Pricing Quality Logistics
Agent Agent Agent   Agent
```

Ara:

- cooperació
- paral·lelisme
- decisions contextuals

Què és un agent?

Entitat software que:

- percep context
- pren decisions
- executa accions
- usa eines
- coopera amb altres agents

Però:

> continua sent software distribuït

Wooldridge, M., & Jennings, N. R. (1995). *Intelligent agents: Theory and practice. The Knowledge Engineering Review, 10(2)*, 115-152.

Orígens històrics del concepte d'Agent

Primers treballs:

- **1973**: Carl Hewitt - **Actor Model**
  (primer model formal d'agents concurrents)
- **1977-1980**: Workshops **DAI**
  (Distributed Artificial Intelligence)
- **1986**: Rodney Brooks - **Subsumption Architecture**
  (arquitectures reactives)
- **1991**: Rao & Georgeff - Model **BDI**
  (Belief-Desire-Intention)

Conferències:

- **1995**: Primera **ICMAS**
  (International Conference on Multi-Agent Systems)
- **2002**: Primera **AAMAS**
  (Autonomous Agents and Multi-Agent Systems)

> Els agents clàssics provenen de la **IA distribuïda** i de la **lògica modal**, no de l'aprenentatge automàtic.

Agents clàssics vs Agents LLM

Multi-agent clàssic

Raonament:

- Lògica modal (Belief-Desire-Intention)
- Sistemes basats en regles
- Planificació simbòlica

Comunicació:

- ACL (Agent Communication Language)
- KQML (Knowledge Query Manipulation Language)

Agents LLM

Raonament:

- Inferència probabilística
- Chain-of-thought prompting
- Reasoning over context

Comunicació:

- Prompts estructurats
- Function calling / Tool use
- Semàntica implícita en LLM

Agents clàssics vs Agents LLM (2)

Multi-agent clàssic

Coneixement:

- Ontologies compartides
- Voc. compartit + definicions formals

Coordinació:

- Contract Net Protocol
- Màquines d'estat finites (FSM)
- Protocols FIPA predefinits: Request, Propose, Dutch Auction...

Agents LLM

Coneixement:

- Embeddings + Vector DB
- Context window dinàmic
- Après durant pretraining

Coordinació:

- Tool orchestration emergent
- Planning amb LLM
- Protocols implícits en prompt

Agents clàssics vs Agents LLM (3)

Multi-agent clàssic

Flexibilitat:

- Baixa: canvis = reescriure regles
- Domini-específic
- Escalabilitat limitada

Observabilitat:

- Alta: traces deterministes
- Debugging precís
- Comportament predictible

Agents LLM

Flexibilitat:

- Alta: adapta per context
- Generalista multi-domini
- Escalabilitat amb cost

Observabilitat:

- Baixa: comportament estocàstic
- Debugging complex (prompt eng.)
- Output no deterministe

La ruptura fonamental

Agents clàssics:

```
if belief("price < threshold"):
    send(PROPOSE(price))
```

Lògica **deductiva**.

Avantatges:

- Verificable formalment
- Comportament garantit
- Traçabilitat completa

Agents LLM:

```
response = llm(
  "Given context X,
   decide optimal action"
)
```

Inferència **probabilística**.

Avantatges:

- Adaptació natural
- Raonament sobre ambigüitat
- Comprensió semàntica rica

> Tesi, es podria dotar els agents LLM de verificabilitat formal? O és inherentment impossible?

Limitacions dels MAS clàssics

Rigidesa de protocols

Contract Net Protocol (Smith, 1980):

```
ANNOUNCE → BID → AWARD → EXECUTE
```

Problema:

- Flux fix i predefinit
- No adapta a situacions noves
- Excepcions requereixen codificar nous estats

FIPA protocols:

- 20+ protocols estandarditzats
- Implementació costosa
- Interoperabilitat limitada en pràctica

Cost de les ontologies

Construcció d'ontologies:

- Experts de domini + enginyers de coneixement
- Mesos/anys per dominis complexos
- Manteniment continu

Problema del mapeig:

```
Ontologia_A ≠ Ontologia_B
```

- Agents de diferents sistemes no comuniquen
- Traducció semàntica complexa
- Mediadors necessaris

Limitacions dels MAS clàssics (2)

Escalabilitat i coordinació

Problema combinatori:

- N agents → N² interaccions potencials
- Protocols de consens costosos
- Blackboard systems: bottleneck central

Contract Net:

- Broadcast a tots els agents
- Overhead de comunicació O(N)
- Decisions centralitzades

Exemple real:

Canviar de 10 a 100 agents pot fer el sistema inviable.

Adaptabilitat zero

Per afegir nova funcionalitat:

1. Redefinir ontologia
2. Reescriure regles de decisió
3. Modificar FSMs
4. Re-testejar tot el sistema
5. Re-desplegar agents

Exemple:

```
Afegir "qualitat ecològica"
a subhasta de peix
```

Requereix:

- Modificar ontologia
- Actualitzar tots els agents
- Recodificar protocols

El problema fonamental

Arquitectura clàssica:

```
Domain Expert
      ↓
  Ontology
      ↓
 Hard-coded rules
      ↓
   FSM protocol
      ↓
  Agent behavior
```

Supòsit:

> El món és completament modelable a priori

Realitat:

- Contextos canvien
- Nous requeriments emergeixen
- Excepcions no previstes
- Domini incomplet

Cost de canvi:

```
High cognitive load
+ Development time
+ Testing effort
+ Deployment risk
```

> Els MAS clàssics eren **fràgils** per disseny.

El nou paradigma: De protocols rígids a semàntics

Abans: Protocol rígid

FIPA Contract Net:

```
1. ANNOUNCE(task)
2. PROPOSE(price, deadline)
3. ACCEPT/REJECT
4. EXECUTE
5. INFORM(result)
```

Característiques:

- Seqüència predefinida
- Missatges tipats estrictament
- Ontologia compartida obligatòria
- Semàntica formal fixada

Ara: Protocol semàntic

LLM-based coordination:

```
Prompt: "Coordina agents per
trobar el millor comprador
considerant risc, preu,
i capacitat logística"
```

Característiques:

- Intent expressat en llenguatge natural
- Adaptació a context
- Semàntica implícita en el model

De ACL a Prompts: L'evolució de la coordinació

Agent Communication Language

FIPA ACL (2002):

```
(REQUEST
 :sender buyer-agent
 :receiver seller-agent
 :content
   (price ?lot123)
 :language FIPA-SL
 :ontology fish-auction
 :protocol fipa-request)
```

Limitacions:

- Ontologia explícita
- Parsing rígid
- Extensibilitat limitada

Semantic Prompts

LLM Tool Calling (2024):

```
{
  "function": "get_best_buyer",
  "arguments": {
    "lot_id": "lot123",
    "constraints": {
      "budget": 5000,
      "risk_tolerance": 0.4,
      "delivery_distance": 50
    }
  }
}
```

Avantatges:

- Schemas flexibles (JSON Schema)
- Raonament contextual
- Zero ontology engineering

Els prompts com a Coordination Contracts

Idea central

Els prompts són:

> **Semantic coordination contracts**

Una evolució de:

```
ACL / RPC contracts
```

Cap a:

```
Intent-based contracts
```

Comparació

ACL Contract:

- Sintaxi rígida
- Semàntica explícita
- Verificable formalment

Prompt Contract:

- Sintaxi flexible
- Semàntica inferida per LLM
- Verificació probabilística

> La coordinació passa de **sintàctica** a **semàntica**

2. Cas d'Estudi: Fish Auction

- Escenari real
- Arquitectura distribuïda
- Pattern Fan-out/Fan-in

Cas d'estudi: Fish Auction Multi-Agent System

Escenari real

Context:

> Arriba un lot de **tonyina premium (200kg)** a la llotja del port de Barcelona a les 06:00h.

Requeriments:

- Decisió en **< 5 minuts**
- Preu competitiu
- Risc financer controlat
- Qualitat verificada
- Logística disponible

Restriccions:

- Múltiples compradors potencials
- Capacitat frigorífica limitada
- Finestra de frescor: 24h

Cas d'estudi: Fish Auction Multi-Agent System

Tasques distribuïdes

Pricing Agent:

- Analitzar preus històrics
- Comparar mercat actual
- Suggerir rang: 4.500€ - 5.200€

Quality Agent:

- Verificar certificació
- Validar temperatura
- Score: 9.2/10

Risk Agent:

- Analitzar historial comprador
- Credit scoring
- Risk score: 0.35 (acceptable)

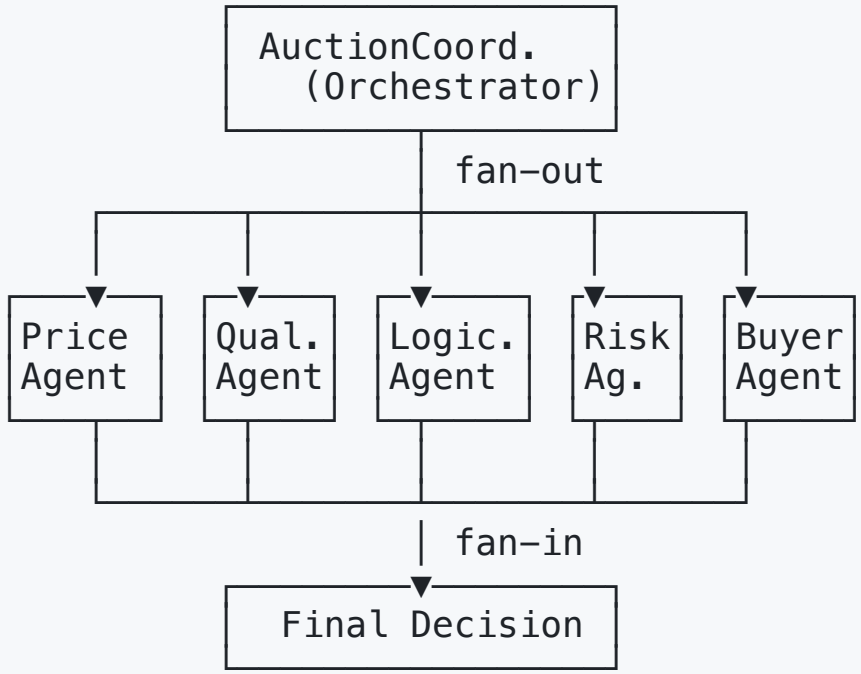
Logistics Agent:

- Disponibilitat camió frigorífic
- Temps lliurament: 3h

Buyer Agent:

- Coordinar i decidir millor oferta

Arquitectura distribuïda del sistema



Flux de coordinació

1. Fan-out paral·lel:

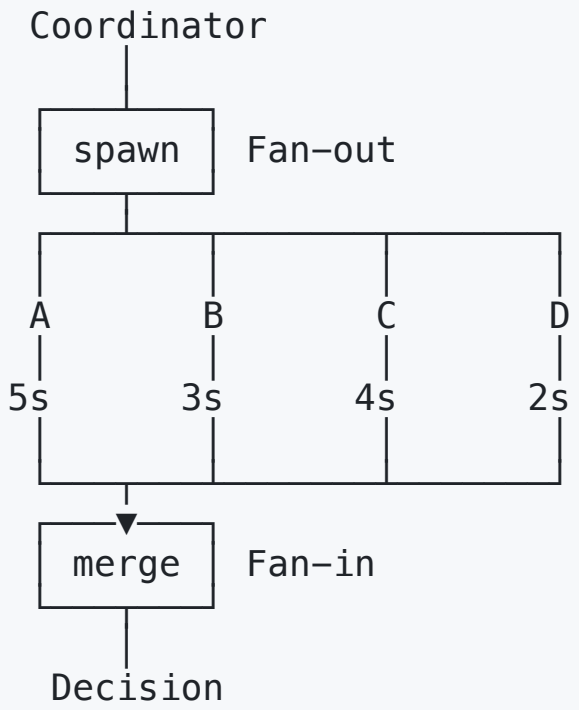
```
tasks = [
  pricing_agent.evaluate(lot),
  quality_agent.inspect(lot),
  logistics_agent.check(lot),
  risk_agent.assess(buyers)
]
results = await asyncio.gather(*tasks)
```

2. Agregació:

```
decision = buyer_agent.decide(
  price=results[0],
  quality=results[1],
  logistics=results[2],
  risk=results[3]
)
```

Pattern: Fan-out / Fan-in

Concepte



Temps total: max(5,3,4,2) = 5s

Implementació

Seqüencial (vell):

```
result_a = agent_a.process()
result_b = agent_b.process()
result_c = agent_c.process()
result_d = agent_d.process()
# Total: 5+3+4+2 = 14s
```

Paral·lel (nou):

```
results = await asyncio.gather(
  agent_a.process(),
  agent_b.process(),
  agent_c.process(),
  agent_d.process()
)
# Total: max(5,3,4,2) = 5s
```

Speedup: 2.8x

Implementació clàssica: MASFIT

Fish Market Multi-Agent System

MASFIT (2004):

> Sistema multi-agent clàssic per gestió de subhastes de peix a mercats reals.

Característiques:

- **FIPA-compliant** agents
- **Contract Net Protocol**
- Ontologia formal del domini
- Arquitectura BDI
- Coordinació explícita

Agents implementats:

- Seller agents
- Buyer agents
- Auctioneer agent
- Information agents

Implementació tècnica

Platform:

- JADE (Java Agent DEvelopment)
- FIPA ACL per comunicació
- Ontologia OWL

Limitacions observades:

- Alta complexitat d'enginyeria
- Ontologia costosa de mantenir
- Adaptació lenta a nous requeriments
- Escalabilitat limitada

Resultat:

> Sistema funcional però **fràgil** davant canvis

Del MASFIT clàssic als sistemes LLM moderns

MASFIT (2004)

```
Ontologia formal
      ↓
FIPA ACL
      ↓
Contract Net
      ↓
Hard-coded rules
```

LLM-based (2024)

```
JSON Schema
      ↓
Function calling
      ↓
Prompt-based coordination
      ↓
LLM reasoning
```

Referència:

> Cuni, G., Esteva, M., Garcia, P., Puertas, E., Sierra, C., & Solchaga, T. (2004). *MASFIT: Multi-Agent System for FIsh Trading*. In *ECAI'04: Proceedings of the 16th European Conference on Artificial Intelligence*.
> PDF: https://www.iiia.csic.es/media/filer_public/3a/e8/3ae8a83c-9f19-4e9e-945a-6b31691b926b/iiia-2004-1182.pdf

Avantatges del paral·lelisme distribuït

Temps de resposta

Seqüencial:

```
T_total = T_a + T_b + T_c + T_d
        = 5 + 3 + 4 + 2
        = 14 segons
```

Paral·lel:

```
T_total = max(T_a, T_b, T_c, T_d)
        = max(5, 3, 4, 2)
        = 5 segons
```

Millora: 180% més ràpid

Patró similar a:

- **Concurrència**: threads/async
- **MapReduce**: map (fan-out) + reduce (fan-in)
- **Scatter-Gather**: microservices
- **Fork-Join**: parallel computing

Diferència clau:

> Ara el **coordinator és un LLM** que decideix dinàmicament:

- Quins agents cridar
- Com agregar respostes
- Què fer amb conflictes

El problema: Loss of Architectural Control

Abans (determinista)

```
def process_auction(lot):
  # Flux explícit
  price = pricing_agent(lot)
  if price > threshold:
    quality = quality_agent(lot)
    if quality > 8:
      logistics = logistics_agent()
      if logistics.available:
        return approve()
  return reject()
```

Control total:

- Ordre fix
- Decisions predefinides
- Debugging trivial

Ara (emergent)

```
def process_auction(lot):
  prompt = f"""Decide best buyer
  for {lot}. Consider: price,
  quality, risk, logistics."""

  # LLM decideix dinàmicament
  decision = llm.decide(
    prompt,
    tools=[pricing, quality,
           logistics, risk]
  )
  return decision
```

Preguntes crítiques:

- Quin agent cridarà primer?
- En quin ordre?
- Quines eines usarà?
- Quan pararà?

Flux: EMERGENT i NO DETERMINISTA

Gaps d'Arquitectura: Necessitem Eines de Control

Les Preguntes Crítiques

? Quin agent cridarà primer?

- Depèn del prompt i raonament del LLM
- **Gap:** No hi ha orquestració explícita
- **Es necessita:** Exposar capacitats disponibles

? En quin ordre?

- Emergeix del context i decisions
- **Gap:** No hi ha flux predefinit
- **Es necessita:** Definir dependències entre eines

? Quines eines usarà?

- El model escull segons necessitat
- **Gap:** Dificultat per limitar accés
- **Es necessita:** Control granular de permisos

? Quan pararà?

- Quan el model "creu" que ha acabat
- **Gap:** Risc de loops infinits o parada prematura
- **Es necessita:** Mecanismes de límits i validació

La Solució: Protocol Estandarditzat

> MCP proporciona l'arquitectura per gestionar aquests gaps mantenint la flexibilitat dels agents

3. Model Context Protocol (MCP)

- Protocol estandarditzat
- Arquitectura MCP
- 4 Gaps arquitectònics i solucions

Model Context Protocol

Definició

Protocol estandarditzat per:

- **Connectar agents amb eines**
- **Exposar capacitats** (capabilities)
- **Gestionar context** compartit
- **Coordinar múltiples models**

Desenvolupat per: Anthropic (2024)

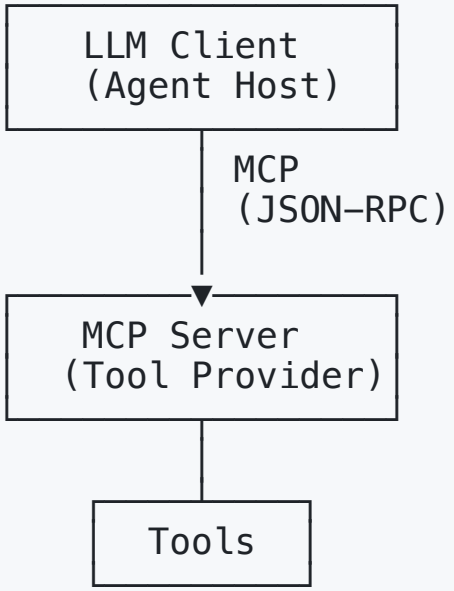
Idea clau:

> "HTTP dels agents, però orientat a capacitats"

Característiques clau

✅ **Service discovery** dinàmic
✅ **Schema validation** (JSON Schema)
✅ **Access control** granular
✅ **Execution limits** configurables
✅ **Error handling** estandarditzat

Arquitectura bàsica



Transport:

- stdio (local)
- HTTP/SSE (remote)

MCP com a Middleware Distribuït

És un **middleware distribuït** amb:

- Service discovery
- Remote procedure calls
- Schema validation
- Error handling
- State management

Equivalències amb SD

Concepte SD	Equivalent MCP
RPC	tool call
Service Registry	tools/list
IDL (WSDL)	JSON Schema
Request/Response	tool invocation
Middleware	MCP Server
Service discovery	resource discovery

Comparació amb RPC tradicional

gRPC / REST:

```
service PricingService {
  rpc GetPrice(PriceRequest)
    returns (PriceResponse);
}
```

MCP:

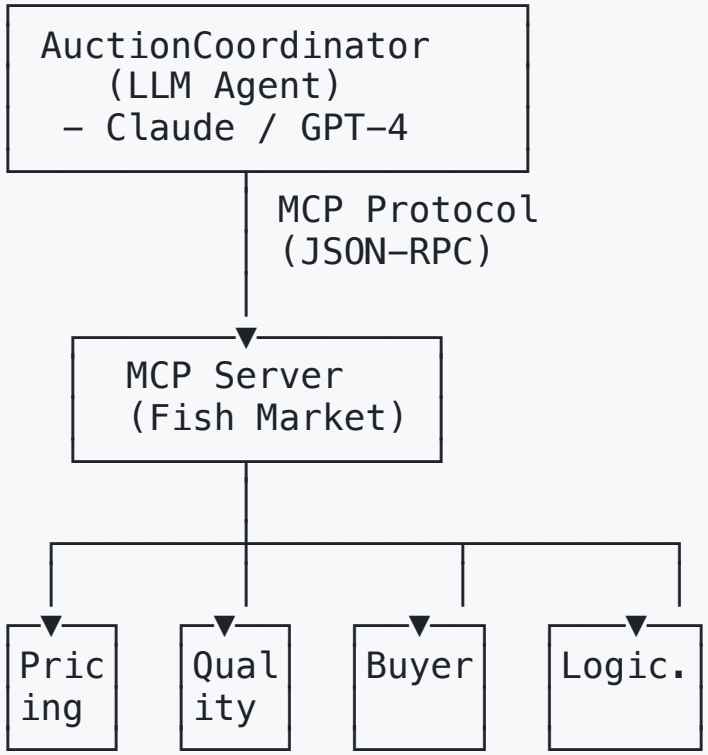
```
{
  "name": "get_market_price",
  "description": "Get current price",
  "inputSchema": {
    "type": "object",
    "properties": {
      "species": {"type": "string"},
      "harbor": {"type": "string"}
    }
  }
}
```

Diferència clau:

> MCP permet **descobriment dinàmic** i **invocació semàntica**

Arquitectura MCP aplicada al Fish Auction

Vista d'arquitectura



Flux d'operacions

1. Discovery:

```
Client → Server: tools/list
Server → Client: [
  "get_market_price",
  "validate_quality",
  "check_buyer_credit",
  "reserve_transport"
]
```

2. Invocation:

```
Client → Server: tools/call
{
  "name": "get_market_price",
  "arguments": {
    "species": "tuna",
    "weight": 200
  }
}
```

3. Response:

```
Server → Client:
{
  "price_per_kg": 12.50,
  "confidence": 0.95
}
```

Evolució del Service Discovery

SOA (2000s):

```
<UDDI>
  <businessEntity>
    <serviceInfo>
      <WSDL>...</WSDL>
    </serviceInfo>
  </businessEntity>
</UDDI>
```

RMI (2010s):

```
// Consul, Eureka, etcd
serviceRegistry.register(
  "pricing-service",
  "http://pricing:8080"
)
```

MCP com a Service Discovery

MCP (2024):

Discovery request:

```
{
  "jsonrpc": "2.0",
  "method": "tools/list",
  "id": 1
}
```

Discovery response:

```
{
  "result": {
    "tools": [
      {
        "name": "get_market_price",
        "description": "Get current market price",
        "inputSchema": {...}
      },
      {
        "name": "validate_quality",
        "description": "Validate fish quality",
        "inputSchema": {...}
      }
    ]
  }
}
```

Avantatge:

> Descobriment **dinàmic** + invocació **semàntica**

De RPC a Tool Calling semàntic

RPC tradicional

Codi fix:

```
# Hard-coded
price = pricing_service.getPrice(
  lot_id="LOT123"
)
```

Característiques:

- Crida **explícita**
- Compilació **estàtica**
- Flux **predeterminat**
- Error si servei no existeix

MCP Tool Calling

Decisió runtime:

```
# LLM decideix què cridar
response = llm.invoke(
  prompt="Evaluate this tuna lot",
  tools=[
    "get_market_price",
    "validate_quality",
    "check_buyer_credit"
  ]
)

# LLM tria: get_market_price
result = mcp_client.call_tool(
  "get_market_price",
  {"species": "tuna", "weight": 200}
)
```

Característiques:

- Crida **dinàmica**
- Descobriment **runtime**
- Flux **emergent**
- Decisió **semàntica**

Cas pràctic: Avaluar un lot amb MCP

Dades del lot

```
{
  "lot_id": "LOT_2024_0513",
  "species": "tuna",
  "quality_grade": "A",
  "weight_kg": 300,
  "current_bid": 12.00,
  "currency": "EUR",
  "arrival_time": "06:30",
  "harbor": "Barcelona"
}
```

Pregunta clau

> És una bona compra per al nostre comprador?

Decisió a prendre

Factors:

- Preu de mercat actual
- Qualitat certificada
- Risc financer del comprador
- Disponibilitat logística
- Marge de profit esperat

Constrains:

```
budget_max = 5000  # €
risk_threshold = 0.4
delivery_time_max = 24  # hores
min_quality_score = 8.0
```

Coordinator Agent: Orquestrador distribuït

Responsabilitats

1. Problem decomposition:

```
tasks = [
  "assess_market_price",
  "verify_quality",
  "check_buyer_risk",
  "validate_logistics"
]
```

2. Tool discovery:

```
available_tools =
  mcp_client.list_tools()

matched_tools =
  match_tasks_to_tools(
    tasks, available_tools
  )
```

3. Parallel coordination:

```
results = await asyncio.gather(
  *[call_tool(t)
    for t in matched_tools]
)
```

4. Result aggregation:

```
aggregated = {
  "price": results[0],
  "quality": results[1],
  "risk": results[2],
  "logistics": results[3]
}
```

5. Decision making:

```
if (aggregated["price"] < threshold
    and aggregated["risk"] < 0.4
    and aggregated["quality"] > 8):
  return "BID"
else:
  return "SKIP"
```

Pattern:

> **Orchestrator distribuït** amb:
>
> Fan-out → Parallel exec → Fan-in

Agent: PricingAgent

Responsabilitat:

> Avaluar preu de mercat i fer recomanació

Tools (MCP):

```
[
  "get_market_price",
  "get_historical_prices",
  "get_competitor_bids"
]
```

Input:

```
{
  "species": "tuna",
  "weight": 300,
  "quality": "A",
  "harbor": "Barcelona"
}
```

Output:

```
{
  "recommended_price": 12.50,
  "market_avg": 12.20,
  "confidence": 0.92,
  "reasoning": "Above avg,
               low volatility"
}
```

Funció en el sistema:

- Analitza preus històrics
- Compara amb competència
- Recomana preu òptim
- Calcula marge de profit

Agent: QualityAgent

Responsabilitat:

> Validar certificacions i qualitat

Tools (MCP):

```
[
  "validate_quality_cert",
  "check_freshness",
  "verify_grade"
]
```

Input:

```
{
  "lot_id": "LOT_2024_0513",
  "quality_grade": "A",
  "arrival_time": "06:30"
}
```

Output:

```
{
  "quality_score": 9.2,
  "cert_valid": true,
  "freshness": "excellent",
  "approved": true
}
```

Funció en el sistema:

- Verifica certificacions
- Valida grau de qualitat
- Comprova frescor del producte
- Aprova o rebutja el lot

Agent: BuyerAgent

Responsabilitat:

> Identificar millor comprador i risc

Tools (MCP):

```
[
  "check_buyer_credit",
  "get_buyer_preferences",
  "calculate_risk_score"
]
```

Input:

```
{
  "buyer_id": "BUYER_042",
  "amount": 3600,
  "species": "tuna"
}
```

Output:

```
{
  "credit_score": 0.85,
  "risk_level": 0.15,
  "payment_terms": "NET30",
  "recommendation": "APPROVE"
}
```

Funció en el sistema:

- Avalua crèdit del comprador
- Calcula risc financer
- Revisa historial de pagaments
- Recomana aprovació o rebuig

Agent: LogisticsAgent

Responsabilitat:

> Validar transport i temps lliurament

Tools (MCP):

```
[
  "reserve_transport",
  "check_delivery_time",
  "verify_cold_chain"
]
```

Input:

```
{
  "weight": 300,
  "destination": "Madrid",
  "required_temp": -18
}
```

Output:

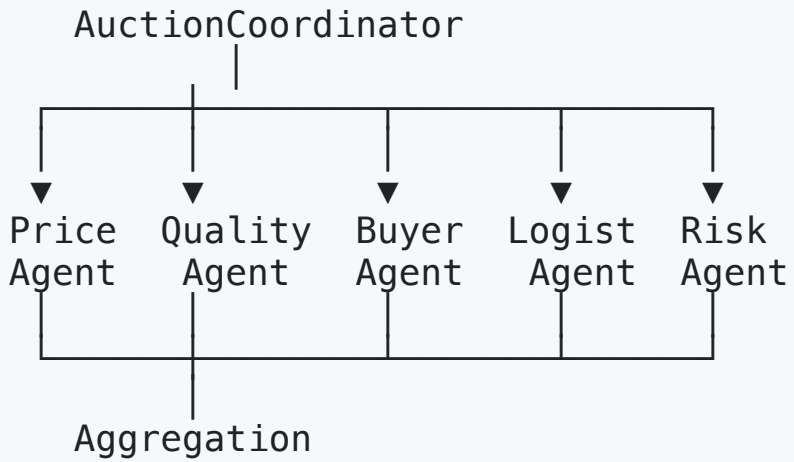
```
{
  "available": true,
  "delivery_hours": 18,
  "cost": 240,
  "cold_chain_certified": true
}
```

Funció en el sistema:

- Verifica disponibilitat transport
- Calcula temps lliurament
- Assegura cadena de fred
- Estima costos logístics

Fan-out paral·lel amb MCP

Pattern distribuït



Analog a:

- Map-Reduce (Map phase)
- Fork-Join pattern
- Scatter-Gather

Implementació amb asyncio

```python
import asyncio

async def evaluate_lot(lot):
  # Fan-out: parallel calls
  results = await asyncio.gather(
    pricing_agent.evaluate(lot),
    quality_agent.evaluate(lot),
    buyer_agent.evaluate(lot),
    logistics_agent.evaluate(lot),
    risk_agent.evaluate(lot),
    return_exceptions=True
  )

  # Fan-in: aggregate
  decision = aggregate_results(
    results
  )

  return decision
```

Beneficis:

- Latency: `max(t1..t5)` vs `sum(t1..t5)`
- Throughput: 4x speedup (cas real)

MCP: Control d'execució i governança

Hem vist com MCP facilita la **coordinació** entre agents.

Ara veurem com MCP proporciona **control arquitectònic**:

Preguntes clau:

1. ❓ **Quin agent cridarà primer?**
   → Service Discovery
2. ❓ **En quin ordre?**
   → Dependencies + Validation
3. ❓ **Quines eines usarà?**
   → Access Control
4. ❓ **Quan pararà?**
   → Execution Limits

Objectiu:

> Proporcionar **governed autonomy**
>
> Els agents tenen flexibilitat per decidir, però dins de límits clarament definits

Analogia SD tradicional:

- Service Discovery (Consul, Eureka)
- Access Control (IAM, RBAC)
- Circuit Breakers
- Rate Limiting

MCP: Dependències i Ordre d'Execució

❓ Pregunta 2: En quin ordre?

Problema:

- Ordre d'execució emergeix del LLM
- No hi ha garanties de precedència
- Dificultat per assegurar constraints

Exemple Fish Auction:

```
Quality check ABANS de pricing
Credit check ABANS d'approval
```

Solució MCP: Metadata i validació

1. Tool metadata:

```
{
  "name": "approve_purchase",
  "inputSchema": {...},
  "requires": [
    "quality_validated",
    "credit_checked"
  ]
}
```

2. Server-side validation:

```
def approve_purchase(buyer_id):
  if not context.has("quality_validated"):
    raise PreconditionFailed(
      "Quality must be validated first"
    )
  if not context.has("credit_checked"):
    raise PreconditionFailed(
      "Credit must be checked first"
    )
  # proceed with approval
```

3. Context tracking:

- MCP Server manté estat de tasques completades
- Valida dependències abans d'executar

MCP: Límits i Terminació

❓ Pregunta 4: Quan pararà?

Problema:

- Agent pot entrar en loops infinits
- Cost d'execució descontrolat
- Risc de timeouts o recursos exhaurits

Exemples perillosos:

```
Agent crida tool → error → retry
→ retry → retry → ∞

Agent "explora" opcions:
tool1 → tool2 → tool3 → ...
→ tool_n → mai acaba
```

Solució MCP: Límits configurables

1. Rate limiting:

```
mcp_server_config = {
  "max_calls_per_minute": 60,
  "max_calls_per_session": 100
}
```

2. Token/cost limits:

```
constraints = {
  "max_tokens": 50000,
  "max_cost_usd": 5.00,
  "timeout_seconds": 300
}
```

3. Circuit breaker pattern:

```
if consecutive_errors > 3:
  circuit_breaker.open()
  raise ServiceUnavailable(
    "Too many errors, stopping"
  )
```

4. Explicit termination signals:

```
{
  "status": "completed",
  "reason": "goal_achieved",
  "final_result": {...}
}
```

Observability: de Request Trace a Reasoning Trace

Microserveis: Request tracing

Distributed tracing (Jaeger, Zipkin):

```
HTTP Request ID: abc123

Service A (10ms)
  └─> Service B (50ms)
      └─> Service C (30ms)
      └─> Service D (20ms)

Total: 110ms
```

Què veiem:

- Latency per servei
- Dependencies chain
- Error propagation
- Bottlenecks

Agents: Reasoning trace

> Com **debugar** i **auditar** decisions d'agents LLM?

```json
{
  "trace_id": "eval_lot_513",
  "lot": {"species": "tuna", "weight": 300},
  "agent": "BuyerAgent",
  "reasoning_steps": [
    {
      "step": 1,
      "action": "call_tool",
      "tool": "get_market_price",
      "input": {"species": "tuna"},
      "result": {"price": 12.50, "avg": 12.20},
      "latency_ms": 120
    },
    {
      "step": 2,
      "action": "llm_reasoning",
      "prompt_tokens": 450,
      "thought": "Price below avg (12.50 < 12.20), favorable"
    },
    {
      "step": 3,
      "action": "call_tool",
      "tool": "check_buyer_credit",
      "result": {"risk": 0.15, "approved": true},
      "latency_ms": 85
    },
    {
      "step": 4,
      "action": "final_decision",
      "output": {
        "decision": "bid",
        "max_bid": 13.00,
        "confidence": 0.88
      }
    }
  ],
  "total_time_ms": 340,
  "model": "gpt-4",
  "cost_usd": 0.0042
}
```

Cas real: Auditoria de decisió errònia

> El sistema va aprovar la compra d'un lot de tonyina a 15€/kg, però el preu de mercat era 12€/kg. Pèrdua: 900€

1. Consultar trace

```
$ grep "lot_2024_0513" logs/
```

```
{
  "trace_id": "lot_2024_0513",
  "timestamp": "2024-05-13T08:30:15Z",
  "final_decision": "BUY",
  "price_paid": 15.00,
  "agents_involved": [
    "PricingAgent",
    "QualityAgent",
    "BuyerAgent"
  ]
}
```

2. Revisar PricingAgent

```
{
  "agent": "PricingAgent",
  "tools_called": [
    "get_market_price"
  ],
  "result": {
    "recommended_price": 12.50
  }
}
```

3. Revisar BuyerAgent

```
{
  "agent": "BuyerAgent",
  "prompt": "Recommend max bid for tuna lot...",
  "context": {
    "market_price": 12.50,
    "quality": "A",
    "buyer_budget": 20000
  },
  "llm_reasoning": "High quality lot, buyer has budget,
                    willing to pay premium",
  "output": {
    "max_bid": 15.00,  // ❌ Massa alt!
    "confidence": 0.92
  }
}
```

Root cause trobat:

- ❌ Prompt no especificava límit de preu vs mercat
- ❌ Agent va prioritzar qualitat sobre preu
- ❌ Cap validació de marge de profit

4. Fix aplicat

Patrons de debugging: Tipus d'errors en agents

1. Errors clàssics (fàcils de debugar)

```
NullPointerException
TimeoutError
ConnectionRefused
JSONDecodeError
```

Característiques:

- Stack trace clar
- Reproducible
- Deterministament
- Fix clar

Exemple Fish Auction:

```
ERROR: Tool 'get_market_price'
       timeout after 5s
```

✅ **Fix:** Incrementar timeout o afegir retry

2. Errors semàntics (difícils de debugar)

```
Decision seems reasonable
BUT is wrong
```

Característiques:

- No hi ha excepció
- Exit code 0 ✅
- Output ben format
- Però decisió incorrecta ❌

Exemple Fish Auction:

```
{
  "agent": "LogisticsAgent",
  "decision": "ship_via_air",
  "reasoning": "Fast delivery important",
  "cost": 1200  // ❌ 10x més car que truck!
}
```

Per què va passar?

- ❓ Prompt no especificava cost constraint
- ❓ Agent va prioritzar velocitat sobre cost

Debugging workflow: 4 nivells d'anàlisi

Nivell 1: Execution trace

Què va passar?

```
{
  "trace_id": "auction_513",
  "agents": ["Pricing", "Quality", "Buyer"],
  "decision": "BUY",
  "final_price": 15.00
}
```

Nivell 2: Tool calls

Quines eines va usar?

```
{
  "agent": "PricingAgent",
  "tools": [
    {
      "name": "get_market_price",
      "input": {"species": "tuna"},
      "output": {"price": 12.50},
      "latency_ms": 120
    }
  ]
}
```

Nivell 3: LLM reasoning

Per què va decidir això?

```
{
  "agent": "BuyerAgent",
  "prompt": "Evaluate lot and recommend...",
  "llm_response": {
    "reasoning": "High quality justifies premium",
    "decision": "bid_15"
  },
  "model": "gpt-4",
  "temperature": 0.7
}
```

Nivell 4: Context & Constraints

Què sabia l'agent?

```
{
  "context": {
    "market_price": 12.50,
    "quality": "A",
    "buyer_budget": 20000
  },
  "constraints": [
    // ❌ FALTA: "max_markup": 1.10
  ]
}
```

Tool grounding: Evitar hallucinations

❌ Problema: Agent inventa dades

Escenari:

```
User: "Quin és el preu de mercat del tonyina?"
Agent: "Crec que ronda els 14€/kg"
```

Risc:

- Agent pot inventar dades
- Hallucination
- Decisions basades en dades falses

Exemple Fish Auction:

```
{
  "agent": "PricingAgent",
  "reasoning": "Based on my knowledge,
               tuna price is around 14€",
  "recommended_price": 14.00
}
```

❌ No hi ha evidència que 14€ sigui correcte

✅ Solució: Tool grounding

Forçar ús de tools:

```
prompt = """
You MUST call get_market_price() tool.
DO NOT guess or estimate prices.
ONLY use data from tools.
"""
```

Output validat:

```
{
  "agent": "PricingAgent",
  "tools_called": [
    {
      "tool": "get_market_price",
      "result": {"price": 12.50, "source": "DB"}
    }
  ],
  "reasoning": "Tool returned 12.50€,
                recommending 13.00€ (markup 1.04)",
  "recommended_price": 13.00,
  "evidence": ["get_market_price_response"]
}
```

✅ Tota decisió està backed by tool data

Fault tolerance: Gestió d'errors en agents

Escenari: Tool timeout

Què passa si:

```
try:
    risk = risk_agent.evaluate()
except Timeout:
    # ❓ Què fem?
```

Opcions:

1. Retry amb backoff

```
@retry(max_attempts=3, backoff=exponential)
def call_risk_agent():
    return risk_agent.evaluate()
```

2. Fallback a default

```
try:
    risk = risk_agent.evaluate()
except Timeout:
    risk = DEFAULT_RISK_MEDIUM
    log.warning("Using default risk")
```

Escenari: Tool retorna error

Exemple:

```
{
  "tool": "get_market_price",
  "error": "External API unavailable",
  "status": 503
}
```

Estratègies:

1. Circuit breaker

```
if consecutive_errors > 3:
    circuit_breaker.open()
    use_cached_price()
```

2. Degraded mode

```
if market_api_down:
    # Usar preus històrics
    price = get_historical_avg(last_7_days)
```

3. Fail-safe amb audit

Auditoria: Què guardar per compliance?

Dades essencials

1. Request & Response

```
{
  "request_id": "lot_513",
  "timestamp": "2024-05-13T08:30:15Z",
  "input": {
    "lot": {"species": "tuna", "weight": 300}
  },
  "output": {
    "decision": "BUY",
    "price": 15.00
  }
}
```

2. Agent execution chain

```
{
  "agents_executed": [
    {"name": "PricingAgent", "duration_ms": 340},
    {"name": "QualityAgent", "duration_ms": 180},
    {"name": "BuyerAgent", "duration_ms": 290}
  ]
}
```

3. Tool calls amb evidència

4. LLM reasoning (crític!)

```
{
  "agent": "BuyerAgent",
  "model": "gpt-4",
  "temperature": 0.7,
  "prompt_template": "buyer_eval_v3",
  "prompt_variables": {
    "market_price": 12.50,
    "quality": "A"
  },
  "llm_output": {
    "reasoning": "High quality, premium justified",
    "decision": "bid_15"
  },
  "tokens_used": 450,
  "cost_usd": 0.0042
}
```

5. Context & Constraints

```
{
  "context": {
    "user_id": "buyer_042",
    "session": "auction_morning",
    "available_budget": 20000
  },
```

Debugging checklist: Preguntes clau

Quan una decisió d'agent va malament, pregunta't:

🔍 Execution level

1. ✅ **Va completar l'execució?**
   - Exit code
   - Excepcions
   - Timeouts
2. ✅ **Tots els agents es van executar?**
   - Trace complet
   - Agents saltats
3. ✅ **Quin va ser l'ordre d'execució?**
   - Seqüencial vs paral·lel
   - Dependencies respectades

🛠️ Tool level

4. ✅ **Quines tools va cridar cada agent?**

🧠 Reasoning level

6. ✅ **Quin prompt va rebre l'agent?**
   - Template usat
   - Variables injectades
   - Constraints explícits
7. ✅ **Quin context tenia disponible?**
   - Tool outputs
   - User inputs
   - System state
8. ✅ **Com va raonar l'LLM?**
   - Reasoning steps
   - Justificació
   - Confidence score

⚙️ System level

Observability + Control = MCP

Hem vist els **reptes d'observability**:

- ❌ Decisions semàntiques errònies
- ❌ Hallucinations sense tool grounding
- ❌ Errors difícils de reproduir
- ❌ Dificultat per auditar raonament

Però també hi ha reptes arquitectònics:

- ❓ Quin agent cridar primer?
- ❓ En quin ordre executar?
- ❓ Quines eines pot usar cada agent?
- ❓ Quan parar l'execució?

MCP proporciona resposta a aquestes 4 preguntes! 🎯

MCP vs Skills: Separació de responsabilitats

Pregunta clau:

> Si MCP gestiona les **eines** (tools), què gestiona el **raonament** dels agents?

MCP: Capa d'Interacció 🔧

Responsabilitat:

> Protocol per **descobrir**, **invocar** i **controlar** eines

MCP NO defineix:

❌ Com **pensa** un agent
❌ Com **raona** un LLM
❌ Què **decideix** fer
❌ Quin **context** usa internament
❌ Quines **heurístiques** aplica

MCP SÍ defineix:

✅ Com **interactua** amb eines
✅ Quins **tools** pot cridar

Skills: Capa de Raonament 🧠

Responsabilitat:

> Coneixement i protocols per **guiar el raonament** de l'agent

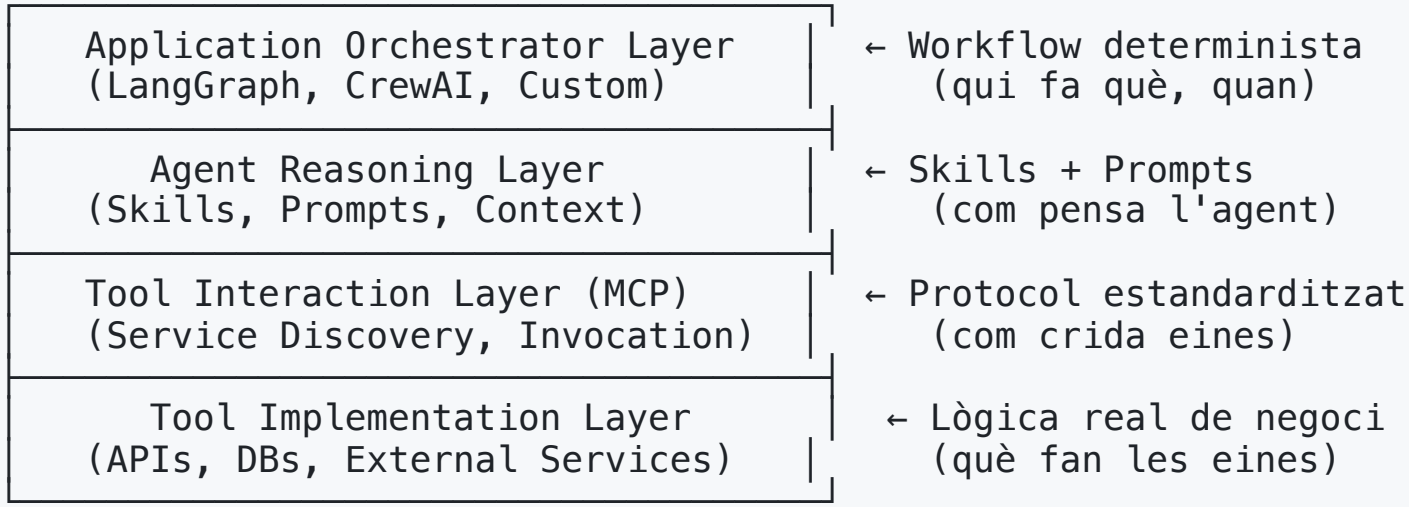
Skills SÍ defineixen:

✅ **Prompt engineering** especialitzat
✅ **Estratègies de decisió**
✅ **Coneixement de domini**
✅ **Heurístiques expertes**
✅ **Protocols de coordinació**
✅ **Context management**

Exemples Fish Auction:

```
BuyerAgent_Skill = {
  "domain": "fish auction",
  "strategy":"minimize risk + maximize profit",
```

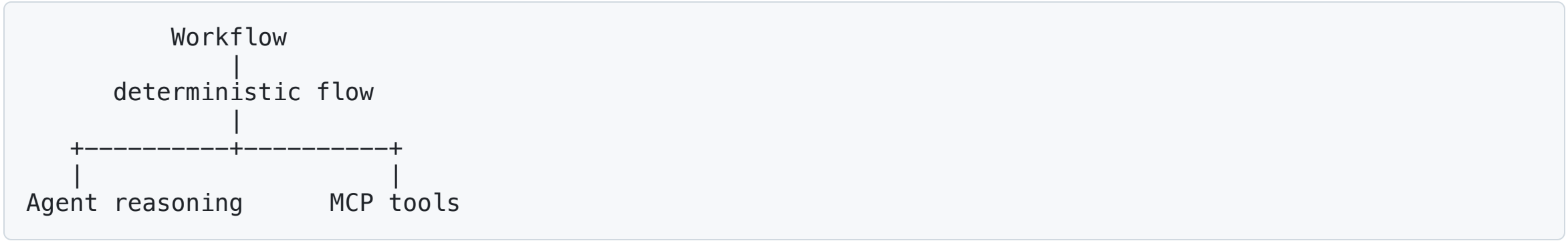
Arquitectura en capes: On va cada cosa?



Principi clau:

> **Workflow** ← deterministc
>
> **Skills** ← intelligence
>
> **MCP** ← interaction
>
> **Tools** ← execution

Patró industrial: Separation of Concerns



Arquitectura de confiança

Regla pràctica:

```
LLM for judgement
Code for control
```

Regla d'or

Useu LLMs per:

- classificació
- decisió contextual
- explicació
- resum
- planificació local

NO per:

- control crític
- permisos
- diners
- autenticació
- arquitectura global

Anti-patterns

❌ Giant God Agent

```
one LLM to rule them all
```

Problemes:

- car
- lent
- impossible de debugar
- poca modularitat

Anti-patterns

❌ Unlimited tools

```
agent can call everything
```

Problemes:

```
tool explosion
```

Anti-patterns

❌ Prompt spaghetti

```
huge prompts
```

Problemes:

```
context pollution
```

Good pattern

✓ Specialized agents

```
PricingAgent
BuyerAgent
RiskAgent
```

petits

explicables

controlables

Good pattern

✓ Structured outputs

NO:

```
maybe looks good
```

SI:

```
{
 "decision":"BUY",
 "confidence":0.88
}
```

Good pattern

✓ Deterministic orchestration

```
workflow fixed
reasoning local
```

Missatge final

Els agents moderns son:

```
Software Distribuït
+
Reasoning
+
Tools
```

La pregunta ja NO és:

```
Com crido un servei?
```

Sinó:

```
Com controlo
un ecosistema d'agents?
```

MCP respon aquesta pregunta:

> Protocol estandarditzat per **descobriment**, **control** i **observabilitat** d'eines en sistemes multi-agent

Següent pas: Com raonen els agents?

Hem vist com **MCP** proporciona:

- ✅ Descobriment de capacitats (Service Discovery)
- ✅ Control d'accés i límits (Governança)
- ✅ Protocol estandarditzat (Interoperabilitat)

Però MCP NO defineix:

- ❌ Com **pensa** un agent
- ❌ Com **raona** sobre decisions
- ❌ Quin **coneixement de domini** usa

Això ho fan els Skills! 🎯

> Els **Skills** injecten coneixement especialitzat i protocols de raonament als agents
>
> Veurem això després del demo pràctic

4. Skills dels Agents

- Què són els Skills
- Diferència entre Skills i Tools
- Arquitectura amb Skills

Protocol rígid vs Protocol semàntic

FIPA ACL (protocol rígid)

FSM determinista:

```
INIT
 ↓
CFP (Call for Proposal)
 ↓
WAIT_PROPOSALS
 ↓
EVALUATE
 ↓
ACCEPT / REJECT
 ↓
CONFIRM
 ↓
DONE
```

Missatge estructurat:

```
(cfp
  :sender buyer
  :receiver supplier
  :content (price tuna 300kg)
  :protocol contract-net
)
```

MCP + LLM (protocol semàntic)

Intent-based constraint:

```
Goal:
  Recommend bid decision

Constraints:
  - budget < 5000€
  - risk ≤ 0.4
  - respect logistics

Tools available:
  - get_market_price
  - validate_quality
  - check_buyer_credit
```

Response (Structured Output):

```
{
  "decision": "bid",
  "max_bid": 4500,
  "confidence": 0.87,
  "reasoning": [
    "Price below market avg",
    "Quality grade A confirmed",
    "Low buyer risk (0.15)"
  ]
}
```

Exemple prompt BuyerAgent

```
Task:
Determine whether a buyer
should bid for this lot.

Buyer profile:
- species=tuna
- max_budget=5000€

Lot:
- tuna
- 300kg
- 12€/kg

Constraints:
- minimize risk
- respect budget

Available tools:
- market_price
- supplier_risk

Return JSON:
{
 "decision":"bid|skip",
 "max_bid":float,
 "reason":string
}
```

Prompts com a Protocols Soft

Prompt NO és text lliure

❌ **Malament:**

```
"Hey, analyze this fish
lot and tell me if it's
good or not"
```

Problemes:

- Resposta imprevisible
- Format inconsistent
- Difícil de processar
- No composable

Prompt ÉS protocol estructurat

✅ **Bé:**

```
{
  "task": "evaluate_lot",
  "context": {
    "lot": {
      "species": "tuna",
      "weight": 300,
      "quality": "A"
    }
  },
  "constraints": [
    "budget < 5000",
    "risk <= 0.4"
  ],
  "output_schema": {
    "decision": "bid|skip",
    "max_bid": "float",
    "confidence": "float",
    "reasoning": "string[]"
  },
  "available_tools": [
    "get_market_price",
    "validate_quality"
  ]
}
```

Concepte clau:

Structured Output: garantir format

Problema: sortida no estructurada

❌ **Text lliure:**

```
"Well, the tuna looks pretty
good, maybe around 13 euros
per kilo would be reasonable,
but I'm not 100% sure..."
```

Impossible:

- Parsejar automàticament
- Validar constrains
- Composar amb altres agents
- Garantir tipus de dades

Solució: JSON Schema

✅ **Sortida validada:**

```
{
  "decision": "bid",
  "max_bid": 13.40,
  "confidence": 0.82,
  "reasoning": [
    "Market price: 12.50€/kg",
    "Quality grade A validated",
    "Buyer risk acceptable (0.15)"
  ],
  "metadata": {
    "evaluation_time": "2024-01-15T08:30:00Z",
    "tools_used": [
      "get_market_price",
      "validate_quality"
    ]
  }
}
```

Benefits:

- Type safety
- Automatic validation
- Composable
- Testable

Tool Schema: IDL per a agents

Definició MCP Tool

```
{
  "name": "get_market_price",
  "description": "Get current market price for fish species",
  "inputSchema": {
    "type": "object",
    "properties": {
      "species": {
        "type": "string",
        "enum": ["tuna", "salmon", "cod"]
      },
      "harbor": {
        "type": "string",
        "description": "Harbor location"
      },
      "weight_kg": {
        "type": "number",
        "minimum": 0
      }
    },
    "required": ["species", "harbor"]
  }
}
```

Comparació amb IDLs tradicionals

gRPC (Protocol Buffers):

```
message PriceRequest {
  string species = 1;
  string harbor = 2;
  float weight_kg = 3;
}

service MarketService {
  rpc GetPrice(PriceRequest)
    returns (PriceResponse);
}
```

OpenAPI (REST):

```
/market/price:
  post:
    parameters:
      - name: species
        in: body
        schema:
          type: string
```

MCP avantatge:

Governed autonomy

No deixem l'agent fer qualsevol cosa.

Exemple:

BuyerAgent:

```
allowed_tools:
- market_price
- supplier_risk
```

NO:

```
filesystem
shell
internet
```

Principle of Least Privilege

Sistemes distribuïts tradicionals

Concepte:

> Donar només els privilegis **mínim necessaris** per complir tasca

Exemples:

```
Unix: user/group permissions
Cloud: IAM roles (AWS, Azure)
K8s: RBAC policies
Docker: capabilities
```

Agents LLM

Bounded autonomy:

```
agent_config = {
  "autonomy": "high",
  "boundaries": {
    "allowed_tools": [
      "read_data",
      "analyze",
      "recommend"
    ],
    "forbidden_actions": [
      "write_db",
      "execute_code",
      "external_api"
    ]
  }
}
```

Principi:

> Agent té **flexibilitat** de decisió, però dins **límits rígids**

Control del flux

Arquitectura industrial real:

```
deterministic backbone
+
probabilistic reasoning
```

No:

```
LLM free for all
```

Example

Flux controlat:

```
Step1 classify
Step2 quality
Step3 pricing
Step4 buyer
Step5 decision
```

L'agent decideix:

```
micro decisions
```

No:

```
macro architecture
```

Què són els Skills dels agents?

Definició

Els **Skills** són:

> Paquets de coneixement especialitzat o capacitats de raonament que s'injecten als agents per augmentar el seu comportament

Exemples de Skills:

- Coneixement de domini específic
- Protocols d'interacció
- Estratègies de decisió
- Best practices
- Heurístiques expertes

Propòsit

✅ Especialitzar comportament dels agents
✅ Injectar coneixement expert
✅ Guiar raonament contextual
✅ Definir protocols d'actuació
✅ Compartir coneixement entre agents

Com funcionen?

1. Injecció via System Prompt:

```
system_prompt = f"""
You are a PricingAgent with skills:

{skill_market_analysis}
{skill_negotiation_strategy}

Use these skills to evaluate...
"""
```

2. Contextualització:

- Skills es carreguen dinàmicament
- Adapten el comportament de l'agent
- Es combinen amb el context del problema

3. Especialització:

```
Base Agent + Skill Set
    → Specialized Agent
```

Skills vs Tools: La diferència clau

Skills (Coneixement)

Què són:

> Coneixement declaratiu i procedural injectat via prompts

Exemples:

```
skill = """
Market Analysis Skill:
- Compare current price with historical avg
- Consider seasonal variations
- Factor in supply/demand indicators
- Apply 10% margin for volatility
"""
```

Característiques:

- ❌ No executen accions
- ✅ Guien raonament
- ✅ Informació passiva
- ✅ Heurístiques i estratègies
- ✅ Context d'expert

Quan usar:

Tools (Accions)

Què són:

> Funcions executables que realitzen accions concretes

Exemples:

```
@mcp.tool()
def get_market_price(species: str):
    return database.query(
        "SELECT price FROM market"
    )
```

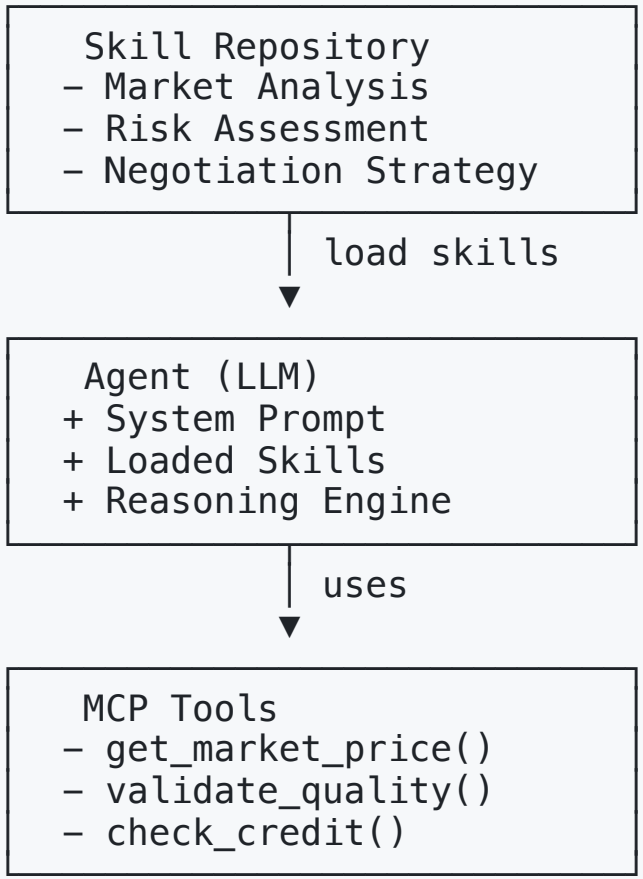
Característiques:

- ✅ Executen accions
- ✅ Retornen dades reals
- ✅ Interactuen amb sistemes
- ❌ No són coneixement
- ✅ Capacitats d'execució

Quan usar:

Arquitectura: Skills + Tools + Agents

Vista arquitectònica



Flux de decisió

1. Agent rep problema:

```
lot = {"species": "tuna", "weight": 300}
```

2. Skills guien raonament:

```
Skill: "Always check historical avg
       before bidding"

Agent thinks: "I should compare with
               historical data first"
```

3. Agent executa Tool:

```
price = call_tool("get_market_price",
                  species="tuna")
```

4. Skills guien decisió:

```
Skill: "Apply 10% margin for volatility"

Agent decides: "Bid = price * 0.90"
```

Exemple pràctic: PricingAgent amb Skills

Skill Definition

```
market_analysis_skill = """
# Market Analysis Skill

## Procedure:
1. Fetch current market price
2. Fetch 30-day historical average
3. Calculate deviation percentage
4. Apply volatility margin (10%)
5. Consider seasonal factors

## Decision Rules:
- If deviation > 20%: HIGH RISK
- If deviation 10-20%: MEDIUM RISK
- If deviation < 10%: LOW RISK

## Bidding Strategy:
- LOW RISK: bid up to market price
- MEDIUM RISK: bid 5% below market
- HIGH RISK: bid 10% below market
"""
```

Agent Implementation

```
class PricingAgent:
    def __init__(self):
        self.skills = [
            market_analysis_skill,
            seasonal_factors_skill
        ]
        self.tools = [
            "get_market_price",
            "get_historical_avg"
        ]

    def evaluate(self, lot):
        system_prompt = f"""
        You are a pricing expert.

        Skills available:
        {self.skills}

        Tools available:
        {self.tools}

        Evaluate: {lot}
        """

        return llm.invoke(
            system_prompt,
            tools=self.tools
        )
```

Resultat:

Skills com a Injecció de Coneixement

El problema

Agents genèrics són ignorants del domini:

```
# Agent base (generic LLM)
agent = Agent(llm="gpt-4")

agent.evaluate(tuna_lot)
# → Generic reasoning
# → No domain expertise
# → Suboptimal decisions
```

La solució: Skill injection

```
# Agent with domain skills
agent = Agent(
    llm="gpt-4",
    skills=[
        market_analysis_skill,
        fish_quality_expertise,
        negotiation_protocols
    ]
)
```

Tipus de Skills

1. Domain Knowledge:

```
- Fish species characteristics
- Market dynamics
- Quality standards
```

2. Procedural Skills:

```
- How to analyze price trends
- How to assess risk
- How to negotiate
```

3. Strategic Skills:

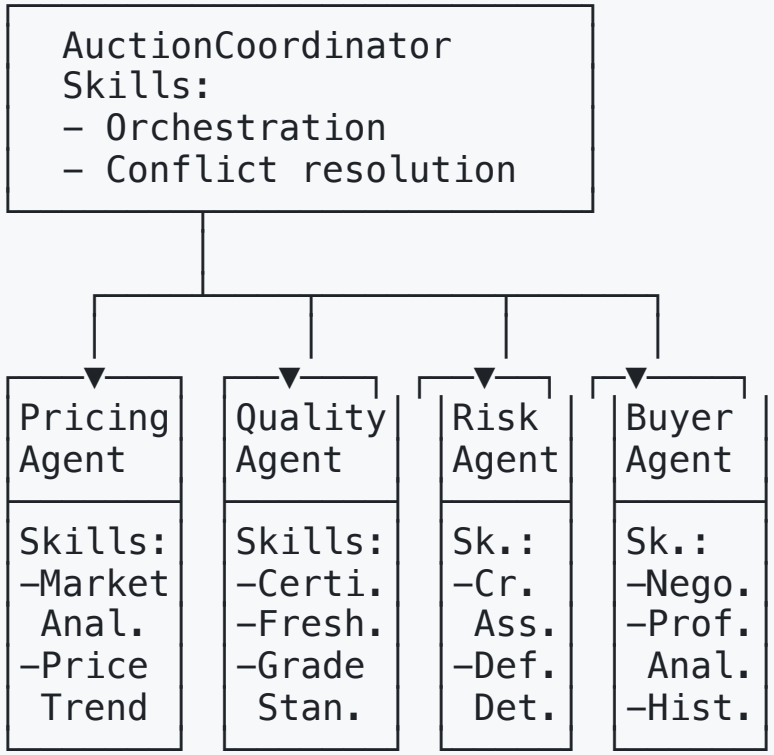
```
- When to bid aggressively
- When to skip
- How to maximize profit
```

4. Protocol Skills:

```
- Interaction patterns
- Communication standards
- Decision workflows
```

Skills en Arquitectures Multi-Agent

Distribució de Skills



Skill Sharing vs Specialization

Shared Skills:

- Tots els agents els tenen
- Ex: Communication protocols
- Ex: Error handling strategies

Specialized Skills:

- Específics per rol
- Ex: PricingAgent → Market Analysis
- Ex: QualityAgent → Certification validation

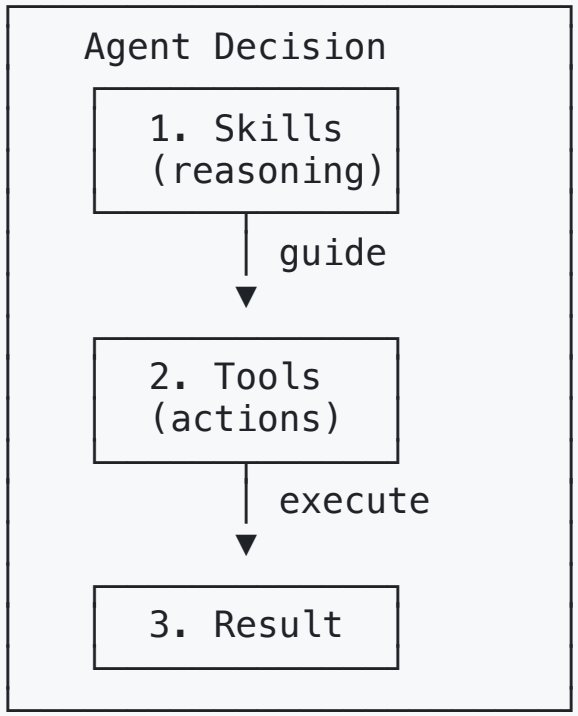
Govern de Skills

Skill Registry:

```
skill_registry = {
    "PricingAgent": [
        "market_analysis_v2.3",
        "price_prediction_v1.5"
    ],
    "QualityAgent": [
        "certification_check_v3.1",
        "freshness_assessment_v2.0"
    ]
```

Relació Skills - MCP Tools

Composició coordinada



Flux:

1. Skill determina **estratègia**
2. Agent tria **tools** adequats
3. Tools obtenen **dades**
4. Skill interpreta **resultats**

Exemple concret

Skill diu:

```
"Before bidding, always verify:
 1. Market price is within 10% of avg
 2. Quality certificate is valid
 3. Buyer credit score > 0.7"
```

Agent executa:

```python
# Step 1: Market check (guided by skill)
market_price = call_tool(
    "get_market_price",
    species="tuna"
)
avg_price = call_tool(
    "get_historical_avg",
    species="tuna", days=30
)

# Skill logic applied
if abs(market_price - avg_price) / avg_price > 0.10:
    return "SKIP: High volatility"

# Step 2: Quality check (guided by skill)
quality = call_tool(
    "validate_quality",
    lot_id=lot_id
)
```

💻 Demo: Fish Auction amb MCP

--

Fitxer: `fish_auction_demo.py`

Arquitectura

```
AuctionCoordinator
        |
   Fan-out paral·lel
        |
   +----+----+----+----+
   |    |    |    |    |
   P    Q    R    L    B
```

- **P**: PricingAgent
- **Q**: QualityAgent
- **R**: RiskAgent
- **L**: LogisticsAgent
- **B**: BuyerAgent

Tools simulats (MCP)

```
get_market_price()
validate_quality()
credit_check()
estimate_transport()
find_best_buyer()
```

Decisió final

```
decision = (
    quality.accepted
    AND logistics.ok
    AND risk < 0.4
    AND buyer_price > bid
)
```

Estructura del codi

1. MCP Tools

```
def get_market_price(species):
    # Simula latency
    time.sleep(random(0.5, 1.5))
    return prices[species]
```

2. Agents especialitzats

```
class PricingAgent:
    def evaluate(self, lot):
        price = get_market_price(
            lot["species"]
        )
        return {
            "recommended_bid": price,
            "confidence": 0.88
        }
```

3. Coordinator

```
class AuctionCoordinator:
    def evaluate(self, lot):
        with ThreadPoolExecutor():
            # Fan-out paral·lel
            p = submit(pricing.evaluate)
            q = submit(quality.evaluate)
            r = submit(risk.evaluate)
            l = submit(logistics.evaluate)
            b = submit(buyer.evaluate)

            # Merge results
            decision = combine(
                p, q, r, l, b
            )
```

🚀 Com executar el demo

Requisits

```
Python 3.7+
```

Cap dependència externa!

Execució

```
cd 11-Agentic
python3 fish_auction_demo.py
```

Output esperat

```
🐟 FISH AUCTION START
📈 [PricingAgent] checking market price
🔍 [QualityAgent] validating quality
⚠️ [RiskAgent] checking risk
🚚 [LogisticsAgent] estimating transport
💰 [BuyerAgent] searching buyer
🐟 [Tool] get_market_price()
🔬 [Tool] validate_quality()
💳 [Tool] credit_check()
🚚 [Tool] estimate_transport()
🛒 [Tool] find_best_buyer()
```

🎯 Conceptes demostrats

Paral·lelisme

```
ThreadPoolExecutor
```

Tots els agents s'executen **simultàniament**

Tool calling

Cada agent crida tools MCP:

```
Agent → Tool → Result
```

Bounded autonomy

Cada agent només veu les seves tools:

```
PricingAgent → market_price
QualityAgent → validate_quality
```

Coordinator determinista

```
decision = combine(
    pricing,
    quality,
    risk,
    logistics,
    buyer
)
```

Flux fix, decisió emergent

Observability

Print statements mostren:

- Ordre d'execució
- Tools cridats
- Resultats intermedis
- Decisió final

5. Implementació Pràctica

- Frameworks (LangChain, LangGraph, CrewAI)
- Local LLM vs API
- Demos funcionals

Stack tecnològic actual (2024-2025)

Frameworks d'agents

LangChain

```
from langchain.agents import initialize_agent
```

- Cadenes + Agents + Tools
- Més genèric, més verbós
- Gran ecosistema

LangGraph

```
from langgraph.graph import StateGraph
```

- **Control explícit del graf**
- Ideal per multi-agent

LLM Runtime

Local

```
ollama run llama3.1
```

- Privacitat
- Cost zero per token
- Latència baixa

API

```
openai.ChatCompletion.create()
anthropic.messages.create()
```

- Millor qualitat
- Sense setup
- Cost per token

LangChain vs LangGraph: Quan usar cada un?

Criteri	LangChain	LangGraph
Use case	Chains lineals, RAG simple	Multi-agent, flux complex
Control	Abstracció alta	Control explícit del graf
Debugging	Difícil	Fàcil (state inspector)
State management	Implícit (memory)	Explícit (StateGraph)
Checkpointing	No	Sí (SQLite, Postgres)
Loops	Limitat	Natiu (cycles en el graf)
Best for	Prototips ràpids	Producció amb complexitat

Regla pràctica:

```
Single agent + linear flow → LangChain
Multi-agent + cycles + state → LangGraph
```

Local LLM vs API: El gran tradeoff

❌ API (OpenAI, Anthropic, etc.)

Avantatges:

- Qualitat superior
- Sense setup
- Sempre actualitzat

Desavantatges:

- Cost per token
- Latència de xarxa
- **Privacitat: les dades surten**
- Rate limits
- Vendor lock-in

✅ Local (Ollama, llama.cpp)

Avantatges:

- **Cost zero** després de download
- **Privacitat total**
- Latència ultra-baixa
- Offline-first

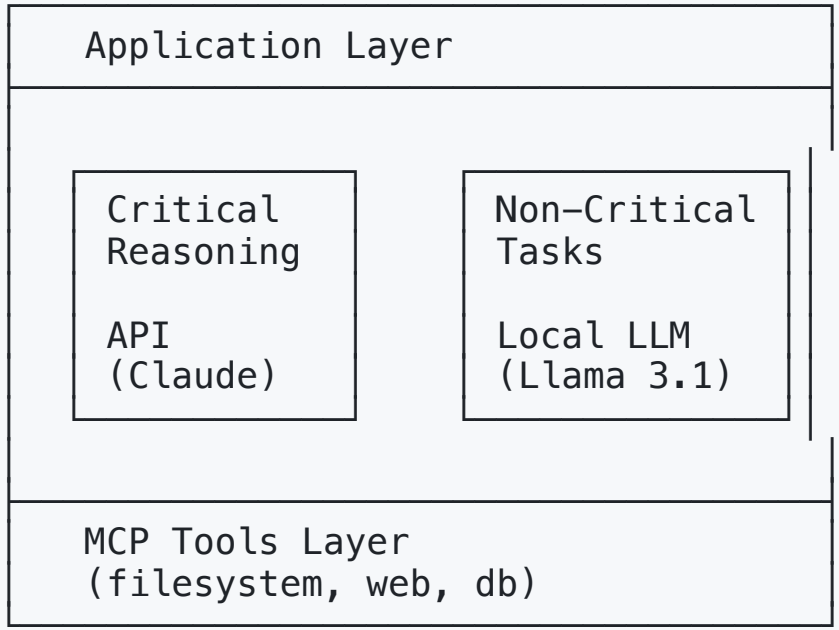
Desavantatges:

- Necessites GPU (o CPU lent)
- Qualitat inferior (però millora)
- Setup inicial

Quan usar:

- Dades sensibles

Arquitectura híbrida recomanada



Estratègia:

- **Tasques crítiques** (diners, legal) → API amb millor model
- **Tasques rutinàries** (resum, classificació) → Local LLM
- **Fallback**: si local falla → API

Exemples d'agents en producció (2025)

Coding Agents

Aider

```
aider --model gpt-4
```

- Edita codi amb git integration
- Multi-file refactoring

Cursor

- IDE amb agent integrat
- Ctrl+K per editar codi
- Multi-agent (composer)

GitHub Copilot Workspace

- Planning → Coding → Testing
- Multi-agent pipeline

Multi-Agent Systems

CrewAI

```
from crewai import Agent, Task, Crew
```

- Agents amb roles
- Tasques seqüencials/paral·leles

AutoGPT / BabyAGI

- Autonomous task decomposition
- Recursive planning

MetaGPT

- Software company simulation
- PM + Architect + Engineer + QA

Tools vs Agents vs CLI: Què triar?

MCP Tool

```python
@mcp.tool()
def get_market_price(species: str):
    return api.fetch(species)
```

Quan:

- Funció determinista
- Input/output clar
- Cap raonament

Exemple: API calls, DB queries

Agent

```
pricing_agent = Agent(
    role="pricing expert",
    tools=[get_market_price]
)
```

Quan:

- Necessites raonament
- Decisió contextual
- Multi-step

Exemple: Avaluar un lot amb context

CLI amb Agent

```
fish-auction-cli evaluate \
  --lot TUNA-302 \
  --model llama3.1
```

Regla d'or

```
Data access → Tool
Reasoning → Agent
User interface → CLI
```

Composició:

```
CLI → Agent → Tools
```

Demo amb LangGraph: Fish Auction

Codi: `fish_auction_langgraph.py`

```python
from langgraph.graph import StateGraph, END
from langchain_ollama import ChatOllama

# Define state
class AuctionState(TypedDict):
    lot: dict
    pricing: Optional[dict]
    quality: Optional[dict]
    decision: Optional[str]

# Build graph
workflow = StateGraph(AuctionState)

workflow.add_node("pricing", pricing_agent)
workflow.add_node("quality", quality_agent)
workflow.add_node("decide", decision_node)

workflow.add_edge("pricing", "decide")
workflow.add_edge("quality", "decide")
workflow.add_edge("decide", END)

app = workflow.compile()
```

Diferència clau: Control explícit del flux amb graf

Comparativa: Demo original vs LangGraph

Original (ThreadPoolExecutor)

```
with ThreadPoolExecutor():
    p = submit(pricing.evaluate)
    q = submit(quality.evaluate)

    pricing = p.result()
    quality = q.result()

    decision = combine(p, q)
```

Pros:

- Simple
- Python estàndard
- Zero dependencies

Cons:

- Sense checkpointing
- Sense time-travel

LangGraph

```
workflow = StateGraph(State)
workflow.add_node("pricing", pricing)
workflow.add_node("quality", quality)
workflow.add_conditional_edges(
    "quality",
    route,
    {"good": "decide", "bad": END}
)

app = workflow.compile(
    checkpointer=MemorySaver()
)

result = app.invoke(
    input,
    config={"thread_id": "123"}
)
```

Pros:

- Checkpointing natiu
- Retry/fallback fàcil
- Depurable (state inspector)

Eines de debugging per agents

LangSmith

```
from langsmith import Client

client = Client()
```

- Trace complet
- Latency breakdown
- Cost tracking
- Playground

Promptfoo

```
prompts:
  - "Evaluate lot {{lot}}"
providers:
  - openai:gpt-4
  - ollama:llama3.1
```

- A/B testing de prompts

OpenLLMetry

```
from traceloop.sdk import Traceloop

Traceloop.init()
```

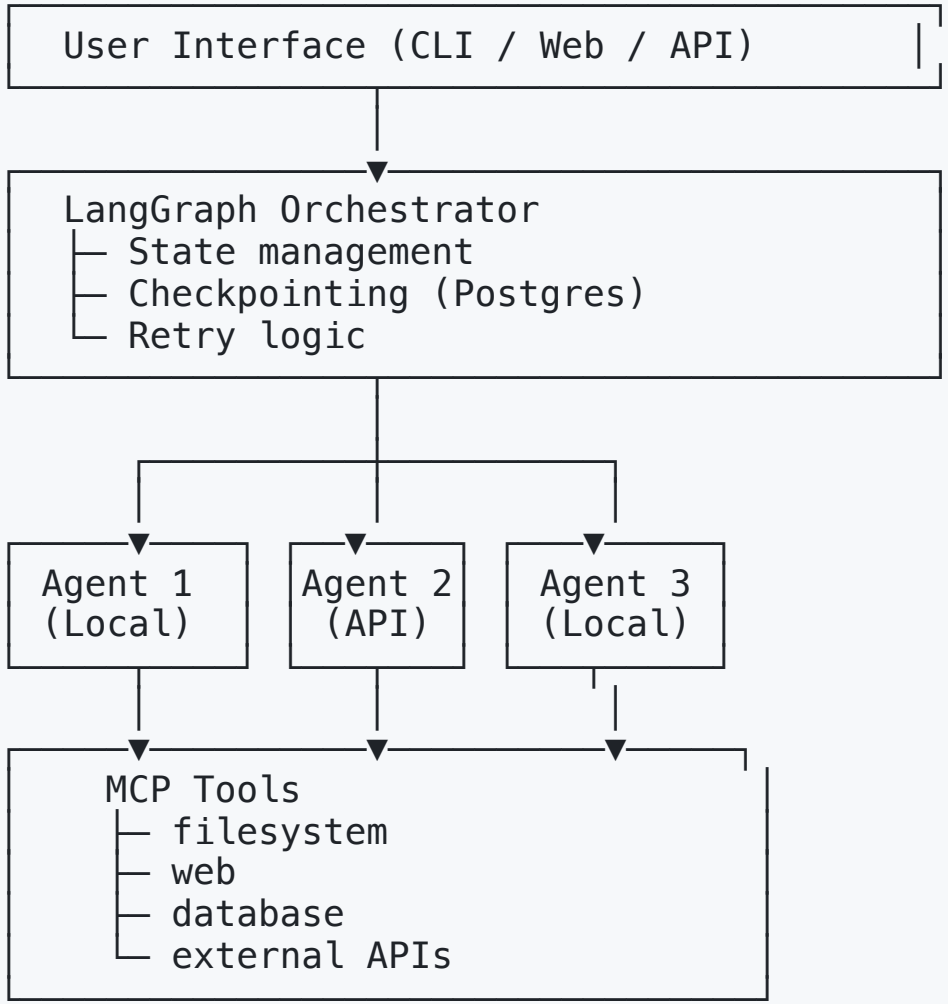
- OpenTelemetry per LLMs
- Integració amb Datadog/Grafana

LangGraph Studio

```
langgraph dev
```

- Visual debugger
- Step-by-step execution
- State inspector
- Time-travel debugging

Arquitectura recomanada per producció



Layers:

1. Interface layer (user-facing)

UV: Gestor de paquets modern per Python

Què és UV?

Gestor de paquets **ultra-ràpid** escrit en Rust

```
# Instal·lació
curl -LsSf https://astral.sh/uv/install.sh | sh
```

10-100x més ràpid que pip

Per què UV?

- ✅ Lock files automàtics (`uv.lock`)
- ✅ Gestió de Python versions
- ✅ Virtual envs automàtics
- ✅ Cache intel·ligent
- ✅ Reproducibilitat total
- ✅ Compatible amb pip/pyproject.toml

Comandes essencials

```
# Setup del projecte
uv sync

# Afegir dependency
uv add langgraph

# Executar sense activar venv
uv run python script.py

# Instal·lar Python
uv python install 3.11

# Actualitzar tot
uv sync --upgrade
```

Per al nostre projecte

```
cd fish-auction/
uv sync --extra langgraph
uv run fish-auction evaluate
```

Setup complet: De zero a producció

1. Instal·lació base

```
# UV (gestor de paquets)
curl -LsSf https://astral.sh/uv/install.sh | sh

# Ollama (LLM local)
curl https://ollama.ai/install.sh | sh
ollama pull llama3.1:8b
```

2. Clone i setup del projecte

```
# Clone repository
git clone <repo> fish-auction
cd fish-auction/

# UV crea venv i instal·la tot automàticament
uv sync --extra all

# O només el que necessites
uv sync --extra langgraph
uv sync --extra crewai
```

3. Executar demos

```
# Demo original (zero dependencies)
```

Materials del curs

Tots els fitxers disponibles al repositori

Fitxers inclosos

Demos Python

fish_auction_demo.py (240 línies)

- Original amb ThreadPoolExecutor
- Zero dependencies

Logging & Observability

auction_logger.py (340 línies)

- Logging estructurat JSON
- Classe `AuctionLogger`
- Classe `LogAnalyzer`

log_analyzer.py (350 línies)

- CLI per anàlisi de logs
- Compare, diff, performance
- 5 subcomandes

logs/ (directori)

- logs/langgraph/
- logs/crewai/

Resum: De teoria a pràctica

1. Conceptes (Slides 1-2700)

- MAS clàssic vs Agents LLM
- Tool calling semàntic
- Governed autonomy
- MCP com a protocol

2. Arquitectura (Slides 2700-2900)

- 4 Gaps arquitectònics
- Solucions MCP
- Observability
- Debugging

3. Implementació (Slides 2900+)

- LangChain vs LangGraph vs CrewAI

Missatge final actualitzat

Els agents moderns son:

```
Software Distribuït
+
Probabilistic Reasoning
+
Tool Ecosystems
```

Les eines ja existeixen:

```
# Setup en 3 comandes
curl -LsSf https://astral.sh/uv/install.sh | sh
ollama pull llama3.1
uv sync --extra langgraph
```

La pregunta ja NO és:

```
És possible?
```

Sinó:

```
Com ho implemento
amb les eines disponibles?
```

6. Observabilitat i Producció

- Logging estructurat
- Debugging d'agents
- Best practices

📊 Observabilitat: El repte del non-determinisme

El problema

```
# Mateix input
lot = {"species": "tuna", ...}

# Execució 1
run_auction(lot)  # → BUY

# Execució 2 (mateix lot!)
run_auction(lot)  # → SKIP ❌
```

Per què passa això?

- 🤖 LLMs són no-deterministes
- 🎲 Temperature > 0 → randomness
- ⚙️ Mateix prompt → diferents outputs

Necessitem

- ✅ Logs estructurats
- ✅ Traces de raonament
- ✅ Comparació de models
- ✅ Detecció d'anomalies
- ✅ Anàlisi de consistència

Sense logging:

- ❌ No saps per què va fallar
- ❌ No pots reproduir bugs
- ❌ No pots comparar models
- ❌ No pots optimitzar

Sistema de logging implementat

auction_logger.py

Mòdul comú per logging estructurat

```
logger = AuctionLogger(
  lot_id="TUNA-302",
  model_name="llama3.1:8b",
  framework="langgraph"
)

logger.log_lot(lot_data)
logger.log_agent_trace(
  "PricingAgent",
  input, output,
  reasoning="Market analysis..."
)
logger.log_tool_call(
  "get_market_price",
  args, result
)
logger.log_final_decision(
  "BUY", reasons, margin
)

logger.save()  # → logs/langgraph/...
```

Què captura?

```
{
  "metadata": {
    "model_name": "llama3.1:8b",
    "timestamp": "2026-05-26T14:30:00"
  },
  "agent_traces": [
    {
      "agent": "PricingAgent",
      "input": {...},
      "output": {...},
      "reasoning": "...",
      "execution_time_seconds": 0.85
    }
  ],
  "tool_calls": [...],
  "final_decision": {
    "decision": "BUY",
    "margin": 0.10
  },
  "execution_time_seconds": 5.23
}
```

Comparació de models: Cas real

Escenari

Vols actualitzar de **llama3.1:8b** a **llama3.2:8b**

Preguntes:

1. Pren les mateixes decisions?
2. És més ràpid?
3. Millora el raonament?

Procés

```
# Executar amb llama3.1:8b
python fish_auction_langgraph_logged.py

# Canviar model i re-executar
MODEL=llama3.2:8b \
  python fish_auction_langgraph_logged.py

# Comparar logs
python log_analyzer.py diff \
```

Output de comparació

```
📊 LOG COMPARISON
==========================================
🔍 Models:
  Model 1: llama3.1:8b
  Model 2: llama3.2:8b

✅ Decisions:
  Model 1: BUY
  Model 2: SKIP
  Match: ❌ NO

⏱️  Execution Times:
  Model 1: 5.23s
  Model 2: 4.87s
  Difference: 0.36s

🧠 Reasoning Differences:
  Agent: DecisionAgent
  Model 1: "Positive margin (0.10)
            and low risk justify BUY"
  Model 2: "Transport delay concerns
            outweigh positive margin"
```

Conclusió: Decisions diferents! Cal

Casos d'ús del logging

1. Debug de raonaments

```
# Lot problemàtic: decisió inesperada
python log_analyzer.py analyze \
  --lot-id TUNA-405

# Revisar reasoning específic
cat logs/.../TUNA-405.json | \
  jq '.agent_traces[3].reasoning'
```

Descobreixes:

"Transport delay of 10h exceeds threshold"

Solució:

Ajustar threshold o millorar prompt

2. Anàlisi de consistència

```
# Executar mateix lot 10 vegades
for i in {1..10}; do
  python fish_auction_langgraph_logged.py
```

Output:

```
📦 Analyzing lot: TUNA-302

✅ Found 10 execution(s)

📊 Summary:
  Decisions: {'BUY': 7, 'SKIP': 3}
  Consistency: 70%
  Average time: 5.12s
```

Interpretació:

✅ >80% consistency → OK

⚠️ 70% consistency → Revisar prompts

❌ <50% consistency → Model inadequat

3. Performance monitoring

```
python log_analyzer.py performance \
  --framework langgraph
```

Identifica:

Estructura de logs i governance

Directori de logs

```
logs/
├── langgraph/
│   ├── 20260526_120000_TUNA-302.json
│   ├── 20260526_130000_SALMON-101.json
│   └── ...
├── crewai/
│   └── (logs de CrewAI)
├── original/
│   └── (versió original)
└── comparisons/
    └── model_comparison.json
```

Naming convention

```
YYYYMMDD_HHMMSS_LOT-ID.json
```

Best practices

✅ DO:

- Loggar SEMPRE en producció
- Comparar models abans de deploy
- Monitorar consistència (target: >80%)
- Revisar logs amb errors
- Arxivar logs històrics
- Usar `log_analyzer.py` per anàlisi

❌ DON'T:

- Ignorar logs amb errors
- Canviar model sense comparar
- Assumir determinisme del LLM
- Esborrar logs sense backup

Eines d'anàlisi: log_analyzer.py

Comandes disponibles

```
# Llistar logs recents
python log_analyzer.py list --limit 10

# Comparar dues execucions
python log_analyzer.py compare log1.json log2.json

# Analitzar tots els logs d'un lot
python log_analyzer.py analyze --lot-id TUNA-302

# Comparar dos models
python log_analyzer.py diff \
  --model1 "llama3.1:8b" \
  --model2 "llama3.2:8b"

# Report de performance
python log_analyzer.py performance --framework langgraph
```

Output exemple: Performance report

```
⚡ PERFORMANCE REPORT
============================================

✅ Analyzing 45 execution(s)

📊 By Framework:
```

Fitxers del sistema de logging

Nous fitxers

auction_logger.py (340 línies)

- Classe `AuctionLogger`
- Classe `LogAnalyzer`
- Logging estructurat JSON
- Mètodes de comparació

fish_auction_langgraph_logged.py (450 línies)

- Versió LangGraph amb logging
- Captura tots els traces
- Guarda automàticament logs
- Compatible amb log_analyzer

Directoris

```
logs/
├── .gitkeep
├── langgraph/
│   └── .gitkeep
├── crewai/
│   └── .gitkeep
├── original/
│   └── .gitkeep
└── comparisons/
    └── .gitkeep
```

.gitignore actualitzat:

```
# Logs (keep structure, ignore content)
logs/**/*.json
!logs/.gitkeep
```