Analiză Detaliată: v4.py - Sistem Multi-AI Ollama

1. Structura Generală a Codului

1.1 Arhitectura Generală

Scriptul `v4.py` este un sistem de conversație multi-AI care orchestrează comunicarea între **N servere Ollama** (minim 2), fiecare cu propriul model AI, rol și configurație. Sistemul funcționează în mod **round-robin**, fiecare agent contribuind la conversație în ordine circulară.

1.2 Componente Principale

```
v4.py
├── Constante globale
├── Funcții utilitare
│   ├── validare_ip()
│   ├── write_to_log()
│   ├── norm_text()
│   └── extrage_fraze_idei()
├── Funcții pentru promptare pe roluri
│   ├── sistem_role_prompt()
│   └── construieste_mesaje()
├── Funcții pentru comunicare API
│   └── api_chat_stream()
└── Funcția main (orchestrator)
```

1.3 Constante Globale

```
PORT_DEFAULT = "11434"              # Port implicit Ollama
MAX_RETRY = 3                        # Număr maxim de reîncercări
LOG_FILE = "conversatie.txt"         # Fișier log pentru conversații
INTREBARE_IMPLICITA = "Ce idei concrete, ne-repetate, poți adăuga mai departe?"
```

---

2. Comunicarea cu Ollama API

2.1 Endpoint și Protocol

Endpoint principal:

```
http://{IP}:{PORT}/api/chat
```

Metoda HTTP: `POST`

Format cerere: JSON

2.2 Structura Payload-ului

```
payload = {
    "model": "qwen3-coder:480b-cloud",  # Numele modelului Ollama
    "messages": [                        # Lista de mesaje (format ChatGPT)
        {"role": "system", "content": "Prompt sistem..."},
        {"role": "user", "content": "Mesaj utilizator..."},
        {"role": "assistant", "content": "Răspuns AI..."}
    ],
    "stream": True,                      # Activează streaming
    "options": {
        "num_predict": 700               # Limită tokeni pentru răspuns
    }
}
```

2.3 Format Răspuns API (Streaming)

Ollama API returnează răspunsuri în format **NDJSON** (Newline Delimited JSON) - câte un JSON pe linie:

```
{"message": {"role": "assistant", "content": "Prima"}, "done": false}
{"message": {"role": "assistant", "content": " parte"}, "done": false}
{"message": {"role": "assistant", "content": " text"}, "done": false}
{"message": {"role": "assistant", "content": ""}, "done": true, "eval_count": 450}
```

Câmpuri importante:

- `message.content` : Fragment de text (token)
- `done` : `true` când răspunsul e complet
- `eval_count` : Număr de tokeni generați (opțional)

2.4 Gestionarea Erorilor și Retry Logic

```
def api_chat_stream(url, payload, timeout):
    retry = 0
    current_timeout = timeout

    while retry < MAX_RETRY:
        try:
            # Trimite request
            resp = requests.post(f"http://{url}/api/chat",
                                 json=payload,
                                 timeout=current_timeout,
                                 stream=True)

            # Procesează streaming
            for line in resp.iter_lines():
                chunk = json.loads(line)
                # Extrage token și afișează

        except requests.exceptions.ReadTimeout:
            retry += 1
            current_timeout *= 2  # Exponential backoff

        except requests.exceptions.RequestException:
            retry += 1
            current_timeout *= 2
```

Caracteristici:
- **Exponential backoff**: Timeout-ul se dublează la fiecare reîncercare
- **Max 3 reîncercări**: După MAX_RETRY, returnează eșec
- **Streaming real-time**: Afișează tokeni pe măsură ce sunt generați

---

3. Gestionarea Rundelor de Conversație

3.1 Flux de Execuție

```
START
  ↓
[Configurare parametri]
  ↓
┌─────────────────────┐
│  RUNDA = 1, 2, 3... │
└─────────────────────┘
  ↓
┌──────────────────────────────────────┐
│  Pentru fiecare mașină (round-robin) │
│  1. Construiește mesaje              │
│  2. Apelează API Ollama              │
│  3. Primește răspuns streaming       │
│  4. Extrage idei noi                 │
│  5. Actualizează istoric conversație │
└──────────────────────────────────────┘
  ↓
[Pauză configurabilă]
  ↓
[Verifică dacă mai sunt runde] → DA → revine la RUNDA
  ↓ NU
END
```

3.2 Parametri Configurabili

```
# Input de la utilizator
subiect = "Tema de discuție"
numar_runde = 5              # 0 = infinit
PAUZA_INTRE_RUNDE_SEC = 3    # Pauză între runde
numar_masini = 2             # Minim 2 mașini
```

3.3 Round-Robin între Agenți

```
for m in masini:  # Fiecare mașină vorbește pe rând
    rol = m["rol"]
    url = m["url"]
    model = m["model"]

    # Construiește context cu istoric + idei
    mesaje = construieste_mesaje(conversatie, subiect, rol, max_tok, idei_list)

    # Apelează API
    rezultat = api_chat_stream(url, payload, timeout)

    # Actualizează conversația comună
    conversatie.append({"role": "assistant", "content": text})
    conversatie.append({"role": "user", "content": INTREBARE_IMPLICITA})
```

Important: Toate mașinile partajează același istoric de conversație (`conversatie`).

---

4. Gestionarea Rolurilor Agenților

4.1 Roluri Disponibile

Rol	Descriere	Comportament Specific
generator	Propune idei noi	Combinații neobișnuite dar plauzibile
critic	Analizează critic	Semnalează riscuri și oferă alternative
moderator	Moderează discuția	Impune noutatea, rezumă, pune întrebări
synth (sintetizator)	Sintetizează	Combină idei, elimină redundanță, creează plan

4.2 Promptare Specifică pe Roluri

```
def sistem_role_prompt(subiect, rol, max_tokens):
    # Bază comună pentru toate rolurile
    baza = f"""Discuți despre: {subiect}.
    Răspunde concis (max {max_tokens} tokeni), clar, în română.

    Reguli generale:
    1) Evită repetițiile
    2) Adaugă cel puțin O IDEE NOUĂ față de lista 'Idei deja menționate'
    3) Listează ideile ca bullets scurte, concrete, testabile
    4) Fii specific (cifre, condiții, costuri, pași)
    """

    # Specializare pe rol
    if rol == "generator":
        return baza + "\nRol: GENERATOR — propune idei noi, combinații neobișnuite"

    if rol == "critic":
        return baza + "\nRol: CRITIC — semnalează riscuri/limitări și oferă îm-
bunătățiri"

    if rol == "moderator":
        return baza + "\nRol: MODERATOR — impune noutatea, rezumă, pune întrebări"

    if rol in ("synth", "sintetizator"):
        return baza + "\nRol: SINTETIZATOR — combină idei, elimină redundanță"
```

4.3 Configurare Default pentru Mașini

```
# Mașină 1 (default)
{
    "ip": "100.76.154.75",
    "port": "11434",
    "model": "qwen3-coder:480b-cloud",
    "rol": "generator",
    "max_tokens": 700,
    "timeout": 120
}

# Mașină 2 (default)
{
    "ip": "100.73.213.29",
    "port": "11434",
    "model": "gpt-oss:120b-cloud",
    "rol": "generator",
    "max_tokens": 700,
    "timeout": 120
}
```

---

5. Structura de Date pentru Mesaje și Conversații

5.1 Lista Mașini (Configurație Agenți)

```
masini: List[Dict[str, Any]] = [
    {
        "url": "100.76.154.75:11434",     # IP:PORT
        "model": "qwen3-coder:480b-cloud", # Nume model Ollama
        "rol": "generator",                # Rol agent
        "max_tokens": 700,                 # Limită tokeni răspuns
        "timeout": 120                     # Timeout inițial (sec)
    },
    # ... alte mașini
]
```

5.2 Istoric Conversație (Comun pentru toți agenții)

```
conversatie: List[Dict[str, str]] = [
    {"role": "system", "content": "Prompt sistem pentru rol..."},
    {"role": "user", "content": "Întrebare/instrucțiune..."},
    {"role": "assistant", "content": "Răspuns AI..."},
    {"role": "user", "content": "Ce idei concrete..."},
    {"role": "assistant", "content": "Noi idei..."},
    # ...
]
```

Roluri posibile:

- `system` : Instrucțiuni pentru AI (rol, reguli)
- `user` : Mesaje de la utilizator sau întrebări implicite
- `assistant` : Răspunsuri de la AI

5.3 Registru de Idei Unice (Deduplicate)

```
registry: Dict[str, bool] = {
    "ideea 1 normalizată lowercase": True,
    "ideea 2 normalizată lowercase": True,
    # ...
}
```

Funcție de extragere idei:

```
def extrage_fraze_idei(text: str) -> List[str]:
    # Împarte pe punct/newline/bullet
    chunks = re.split(r'[.\n•\-\u2022;]+', text)

    # Normalizează și filtrează
    out = []
    for c in chunks:
        t = norm_text(c)  # lowercase + strip whitespace
        if len(t) >= 5:    # Ignoră fraze prea scurte
            out.append(t)
    return out
```

5.4 Construirea Mesajelor pentru API

```
def construieste_mesaje(conversatie, subiect, rol, max_tokens, idei_deja):
    # 1. Prompt sistem cu rol specific
    sistem = sistem_role_prompt(subiect, rol, max_tokens)

    # 2. Lista idei deja menționate (ultimele 30)
    idei_block = "Idei deja menționate:\n- " + "\n- ".join(idei_deja[-30:])

    # 3. Instrucțiune de noutate
    instructiune = f"{idei_block}\n" \
                   "Asigură-te că fiecare punct nou este ne-repetat. " \
                   f"După idei, încheie cu: 'NEXT: {INTREBARE_IMPLICITA}'"

    # 4. Construiește lista finală
    messages = [{"role": "system", "content": sistem}]
    messages.extend(conversatie)  # Istoric complet
    messages.append({"role": "user", "content": instructiune})

    return messages
```

---

6. Funcționalități Importante de Păstrat

6.1 Sistem de Noutate (Anti-Repetare)

Componentă critică pentru calitatea conversației:

1. **Registru global de idei**: Toate ideile menționate sunt stocate normalizate
2. **Injectare în prompt**: Ultimele 30 idei sunt trimise fiecărui agent
3. **Verificare la runtime**: Numărul de idei noi adăugate e tracked

```
# După fiecare răspuns
fraze = extrage_fraze_idei(text)
adaugate = 0
for fr in fraze:
    if fr not in registry:
        registry[fr] = True
        adaugate += 1

print(f"Idei noi adăugate: {adaugate}")
```

6.2 Logging Detaliat

Toate interacțiunile sunt loggate în `conversatie.txt`:

```
def write_to_log(message: str):
    with open(LOG_FILE, "a", encoding="utf-8") as f:
        f.write(message + "\n")

# Usage
write_to_log(f"[{rol.upper()} @ {url} :: tokens={tokens}] {text}")
write_to_log(f"[INFO] Idei noi adăugate: {adaugate}")
```

6.3 Validare IP

```
def validare_ip(ip_str: str) -> bool:
    pattern = r'^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4]
[0-9]|[01]?[0-9][0-9]?)$'
    return re.match(pattern, ip_str) is not None
```

6.4 Retry Logic cu Exponential Backoff

```
retry = 0
current_timeout = timeout

while retry < MAX_RETRY:
    try:
        # Încearcă request
        # ...
    except requests.exceptions.ReadTimeout:
        retry += 1
        current_timeout *= 2  # Exponential backoff: 120s → 240s → 480s
```

6.5 Streaming Real-Time

Afișarea token-by-token pentru feedback instant:

```
for line in resp.iter_lines():
    chunk = json.loads(line)
    if "message" in chunk:
        token = chunk["message"]["content"]
        raspuns_complet += token
        sys.stdout.write(token)  # Afișare instant
        sys.stdout.flush()
```

6.6 Flexibilitate Configurație

- **N mașini** (>=2): Suport pentru orice număr de agenți
- **Modele diferite**: Fiecare agent poate folosi alt model
- **Limite tokeni personalizate**: Per agent
- **Timeout-uri diferite**: Per agent
- **Roluri mixte**: Combinație liberă de roluri

---

7. Recomandări pentru Integrarea în FastAPI

7.1 Arhitectură Propusă

```
FastAPI App
├── Modele Pydantic
│   ├── MachineConfig (configurație agent)
│   ├── ConversationMessage (mesaj)
│   └── ConversationState (stare conversație)
├── Endpoints REST
│   ├── POST /api/conversations/start
│   ├── POST /api/conversations/{id}/round
│   ├── GET /api/conversations/{id}/history
│   └── WebSocket /ws/conversations/{id}
├── Services
│   ├── OllamaAPIService (comunicare cu Ollama)
│   ├── ConversationManager (orchestrare runde)
│   └── IdeaRegistryService (management idei)
└── Database
    ├── conversations (sesiuni conversații)
    ├── messages (istoric mesaje)
    └── ideas (registru idei)
```

7.2 Componente de Reutilizat Direct

Funcție Original	Destinație în FastAPI	Modificări Necesare
`api_chat_stream()`	`OllamaAPIService.chat_stream()`	Întoarce generator async
`sistem_role_prompt()`	`PromptBuilder.build_system_prompt()`	Fără modificări
`construieste_mesaje()`	`PromptBuilder.build_messages()`	Fără modificări
`extrage_fraze_idei()`	`IdeaExtractor.extract_phrases()`	Fără modificări
`validare_ip()`	`Validators.validate_ip()`	Folosește în Pydantic

7.3 Modificări Cheie

A. Async/Await pentru FastAPI

```
# Original (sync)
def api_chat_stream(url, payload, timeout):
    resp = requests.post(...)
    for line in resp.iter_lines():
        yield chunk

# FastAPI (async)
import httpx

async def api_chat_stream(url, payload, timeout):
    async with httpx.AsyncClient() as client:
        async with client.stream("POST", ...) as resp:
            async for line in resp.aiter_lines():
                yield chunk
```

B. WebSocket pentru Streaming UI

```
@app.websocket("/ws/conversations/{conversation_id}")
async def websocket_endpoint(websocket: WebSocket, conversation_id: str):
    await websocket.accept()

    async for token in ollama_service.chat_stream(...):
        await websocket.send_json({
            "type": "token",
            "content": token,
            "timestamp": datetime.now().isoformat()
        })

    await websocket.send_json({"type": "done"})
```

C. Persistență Date (PostgreSQL/SQLite)

```
class Conversation(Base):
    __tablename__ = "conversations"

    id = Column(String, primary_key=True)
    subject = Column(String)
    created_at = Column(DateTime)
    status = Column(String)  # "active", "completed", "paused"

class Message(Base):
    __tablename__ = "messages"

    id = Column(Integer, primary_key=True)
    conversation_id = Column(String, ForeignKey("conversations.id"))
    role = Column(String)  # "system", "user", "assistant"
    content = Column(Text)
    agent_url = Column(String, nullable=True)
    tokens = Column(Integer, nullable=True)
    created_at = Column(DateTime)
```

7.4 Endpoints Esențiali

```
# 1. Pornire conversație nouă
POST /api/conversations/start
{
    "subject": "Tema discuție",
    "machines": [
        {
            "url": "100.76.154.75:11434",
            "model": "qwen3-coder:480b-cloud",
            "role": "generator",
            "max_tokens": 700,
            "timeout": 120
        }
    ],
    "num_rounds": 5,
    "pause_seconds": 3
}

# 2. Execută o rundă
POST /api/conversations/{id}/round
Response: {
    "round": 3,
    "messages": [...],
    "new_ideas_count": 5
}

# 3. Istoric complet
GET /api/conversations/{id}/history
Response: {
    "conversation": {...},
    "messages": [...],
    "ideas": [...]
}

# 4. Streaming live (WebSocket)
WS /ws/conversations/{id}
```

---

8. Dependențe și Cerințe

8.1 Dependențe Python

```
requests==2.31.0       # HTTP client (sau httpx pentru async)
pydantic==2.5.0        # Validare date
fastapi==0.104.1       # Web framework
uvicorn==0.24.0        # ASGI server
sqlalchemy==2.0.23     # ORM pentru DB
httpx==0.25.0          # Async HTTP client
```

8.2 Cerințe Sistem

- **Python**: 3.8+
- **Ollama**: Instalat pe fiecare mașină server
- **Port**: 11434 (default Ollama) trebuie deschis
- **Rețea**: Acces între servere (IP-uri accesibile)

8.3 Modele Ollama Menționate

```
# Modele folosite în configurația default
ollama pull qwen3-coder:480b-cloud
ollama pull gpt-oss:120b-cloud
ollama pull llama3:latest
```

---

9. Puncte de Atenție pentru Implementare

9.1 ⚠️ Probleme Potențiale

1. **Timeout-uri**: Modelele mari (120b-480b) pot fi lente
   - **Soluție**: Timeout-uri adaptive, progres indicator în UI

2. **Memorie conversație**: Istoricul crește necontrolat
   - **Soluție**: Limitare la ultimele N mesaje (ex: 50)

3. **Concurență**: Multiple conversații simultane
   - **Soluție**: Task queue (Celery/RQ) sau async workers

4. **Erori rețea**: Servere offline sau inaccesibile
   - **Soluție**: Health checks, fallback la servere alternative

9.2 ✅ Best Practices

1. **Cache răspunsuri**: Pentru întrebări repetate
2. **Rate limiting**: Protecție împotriva abuzului
3. **Monitoring**: Logare structurată (JSON logs)
4. **Autentificare**: JWT tokens pentru API
5. **Validare input**: Sanitizare subiect, limite configurare

---

10. Rezumat Flux de Date

```
[USER INPUT]
    ↓
[Configurare: subiect, mașini, runde]
    ↓
[Conversație Loop]
    ↓
┌─────────────────────────────────┐
│  Pentru fiecare agent:          │
│  1. Construire context          │ → [registry idei] + [istoric conversație]
│  2. API call Ollama             │ → Streaming tokens
│  3. Extragere idei noi          │ → Actualizare registry
│  4. Salvare în istoric          │ → conversatie.append()
└─────────────────────────────────┘
    ↓
[Pauză]
    ↓
[Rundă următoare sau Sfârșit]
    ↓
[LOG FILE + Console Output]
```

---

11. Checklist Migrare la Web

- [ ] Convertire funcții sync → async
- [ ] Înlocuire `requests` cu `httpx`
- [ ] Creare modele Pydantic pentru validare
- [ ] Setup database (SQLAlchemy models)
- [ ] Implementare endpoints REST
- [ ] Implementare WebSocket pentru streaming
- [ ] UI React/Vue pentru interfață vizuală
- [ ] Sistem autentificare utilizatori
- [ ] Persistență conversații în DB
- [ ] Export conversații (JSON, PDF, TXT)
- [ ] Dashboard monitoring (tokeni, cost, erori)
- [ ] Teste unitare și integrare
- [ ] Documentație API (Swagger/OpenAPI)

Data analiză: 23 octombrie 2025
Fișier analizat: `/home/ubuntu/Uploads/v4.py`
Versiune: v4 (multi-mașini cu roluri)