**Addendum: Enhanced Ollama Integration & Streaming Architecture**

**Overview**

The additional Ollama documentation reveals **production-ready streaming capabilities and advanced configuration management** that significantly enhances the Project Citadel architecture. These components provide critical infrastructure for real-time AG-UI integration and sophisticated model parameter tuning.

**1. Enhanced Streaming Architecture Integration**

**1.1 Advanced Streaming Capabilities**

The additional documentation reveals sophisticated streaming infrastructure that perfectly aligns with AG-UI requirements:

python

# Enhanced streaming with print control and memory management

class ProductionOllamaClient(OllamaRESTClient):

"""Production-enhanced Ollama client with advanced streaming"""

def \_\_init\_\_(self, model: str = "deepcoder:14b", host: str = "http://localhost:11434"):

super().\_\_init\_\_(model, host)

self.response\_buffer = []

self.streaming\_stats = {"tokens\_streamed": 0, "response\_time": 0}

async def generate\_stream\_for\_agui(

self,

prompt: str,

model\_override: str = None,

temperature: float = 0.7,

top\_p: float = 0.9,

repeat\_penalty: float = 1.0,

print\_stream: bool = False # Disable console output for AG-UI

) -> AsyncIterator[Dict[str, Any]]:

"""Enhanced streaming specifically for AG-UI integration"""

start\_time = time.time()

model = model\_override or self.model

# Use the proven streaming implementation

url = f"{self.base\_url}/api/generate"

payload = {

"model": model,

"prompt": prompt,

"stream": True,

"temperature": temperature,

"top\_p": top\_p,

"repeat\_penalty": repeat\_penalty

}

async with aiohttp.ClientSession() as session:

async with session.post(url, json=payload) as response:

if response.status != 200:

raise RuntimeError(f"Streaming Error: {response.status}")

token\_count = 0

async for line in response.content:

if line:

try:

data = json.loads(line.decode('utf-8'))

token = data.get("response", "")

if token:

token\_count += 1

# Yield formatted data for AG-UI

yield {

"content": token,

"token\_count": token\_count,

"model\_used": model,

"temperature": temperature,

"timestamp": time.time()

}

except json.JSONDecodeError:

continue

# Final statistics

self.streaming\_stats = {

"tokens\_streamed": token\_count,

"response\_time": time.time() - start\_time,

"model\_used": model

}

**1.2 FastAPI Streaming Integration for AG-UI**

python

# Enhanced FastAPI server with AG-UI optimized endpoints

from fastapi import FastAPI, Request

from fastapi.responses import StreamingResponse

from pydantic import BaseModel

from typing import Optional

import asyncio

import json

app = FastAPI(title="Citadel Ollama Gateway", version="2.1.0")

class EnhancedGenerateRequest(BaseModel):

prompt: str

model: Optional[str] = "deepseek-r1:latest"

stream: bool = True

use\_memory: bool = False

temperature: float = 0.7

top\_p: float = 0.9

repeat\_penalty: float = 1.0

max\_tokens: Optional[int] = None

# AG-UI specific parameters

include\_stats: bool = True

chunk\_delay\_ms: int = 0 # For rate limiting if needed

class CitadelOllamaGateway:

"""Production gateway with enhanced streaming for AG-UI"""

def \_\_init\_\_(self):

self.clients = {} # Model-specific clients

self.default\_models = {

"fast": "mistral:latest",

"balanced": "deepseek-r1:latest",

"code": "deepcoder:14b",

"premium": "deepseek-r1:32b"

}

def get\_client(self, model: str) -> ProductionOllamaClient:

"""Get or create client for specific model"""

if model not in self.clients:

self.clients[model] = ProductionOllamaClient(model=model)

return self.clients[model]

async def stream\_generate(self, request: EnhancedGenerateRequest):

"""Stream generation optimized for AG-UI consumption"""

client = self.get\_client(request.model)

async for chunk in client.generate\_stream\_for\_agui(

prompt=request.prompt,

temperature=request.temperature,

top\_p=request.top\_p,

repeat\_penalty=request.repeat\_penalty,

print\_stream=False # Never print to console in production

):

# Format for Server-Sent Events (AG-UI compatible)

if request.include\_stats:

chunk["stats"] = client.streaming\_stats

# Rate limiting if requested

if request.chunk\_delay\_ms > 0:

await asyncio.sleep(request.chunk\_delay\_ms / 1000)

yield f"data: {json.dumps(chunk)}\n\n"

# Send completion marker

yield f"data: {json.dumps({'type': 'completion', 'stats': client.streaming\_stats})}\n\n"

# Initialize gateway

gateway = CitadelOllamaGateway()

@app.post("/api/generate/stream")

async def stream\_generate\_endpoint(request: EnhancedGenerateRequest):

"""Streaming endpoint optimized for AG-UI"""

return StreamingResponse(

gateway.stream\_generate(request),

media\_type="text/event-stream",

headers={

"Cache-Control": "no-cache",

"Connection": "keep-alive",

"Access-Control-Allow-Origin": "\*",

"Access-Control-Allow-Headers": "\*",

}

)

@app.post("/api/generate")

async def generate\_endpoint(request: EnhancedGenerateRequest):

"""Non-streaming endpoint for compatibility"""

client = gateway.get\_client(request.model)

# Use the existing non-streaming generation

result = client.generate(

prompt=request.prompt,

use\_memory=request.use\_memory,

temperature=request.temperature,

top\_p=request.top\_p,

repeat\_penalty=request.repeat\_penalty

)

return {

"response": result,

"model": request.model,

"stats": client.streaming\_stats

}

@app.get("/api/models")

async def list\_models():

"""List available models with capabilities"""

return {

"available\_models": list(gateway.default\_models.values()),

"model\_types": gateway.default\_models,

"default\_model": gateway.default\_models["balanced"]

}

@app.get("/api/health")

async def health\_check():

"""Health check with model status"""

model\_status = {}

for model\_type, model\_name in gateway.default\_models.items():

try:

client = gateway.get\_client(model\_name)

# Quick health check

test\_result = client.generate("Hello", temperature=0.1)

model\_status[model\_name] = {

"status": "healthy",

"type": model\_type,

"last\_response\_length": len(test\_result)

}

except Exception as e:

model\_status[model\_name] = {

"status": "error",

"type": model\_type,

"error": str(e)

}

return {

"service\_status": "healthy",

"models": model\_status

}

**2. Advanced Configuration Management**

**2.1 Dynamic Parameter Injection**

The configuration injection capabilities provide sophisticated model tuning:

python

# Enhanced configuration management for optimal performance

class ConfigurationManager:

"""Advanced configuration management for different use cases"""

def \_\_init\_\_(self):

self.optimization\_profiles = {

"creative\_writing": {

"temperature": 0.9,

"top\_p": 0.95,

"repeat\_penalty": 1.1,

"description": "High creativity for content generation"

},

"technical\_analysis": {

"temperature": 0.3,

"top\_p": 0.8,

"repeat\_penalty": 1.0,

"description": "Low temperature for factual accuracy"

},

"code\_generation": {

"temperature": 0.1,

"top\_p": 0.9,

"repeat\_penalty": 1.05,

"description": "Precise code generation"

},

"conversational": {

"temperature": 0.7,

"top\_p": 0.9,

"repeat\_penalty": 1.0,

"description": "Balanced conversation"

},

"summarization": {

"temperature": 0.4,

"top\_p": 0.85,

"repeat\_penalty": 1.1,

"description": "Focused summarization"

}

}

def get\_optimal\_config(

self,

task\_type: str,

content\_type: str = None,

user\_preferences: Dict[str, Any] = None

) -> Dict[str, float]:

"""Get optimal configuration for specific task"""

# Start with base profile

if task\_type in self.optimization\_profiles:

config = self.optimization\_profiles[task\_type].copy()

else:

config = self.optimization\_profiles["conversational"].copy()

# Adjust based on content type

if content\_type == "academic\_paper":

config["temperature"] \*= 0.8 # More conservative

config["top\_p"] \*= 0.9

elif content\_type == "creative\_content":

config["temperature"] \*= 1.2 # More creative

config["top\_p"] \*= 1.05

# Apply user preferences

if user\_preferences:

for key, value in user\_preferences.items():

if key in config and isinstance(value, (int, float)):

config[key] = max(0.0, min(2.0, float(value))) # Clamp values

# Remove description for API usage

config.pop("description", None)

return config

# Integration with existing LLM manager

class EnhancedLLMManager(LLMManager):

"""LLM Manager with advanced configuration support"""

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.config\_manager = ConfigurationManager()

self.performance\_cache = {}

async def generate\_with\_optimal\_config(

self,

prompt: str,

task\_type: str,

content\_type: str = None,

model\_name: str = None,

user\_preferences: Dict[str, Any] = None,

stream: bool = False

):

"""Generate text with automatically optimized configuration"""

# Get optimal configuration

optimal\_config = self.config\_manager.get\_optimal\_config(

task\_type=task\_type,

content\_type=content\_type,

user\_preferences=user\_preferences

)

# Auto-select model if not specified

if not model\_name:

model\_name = await self.auto\_select\_model(prompt, task\_type)

# Generate with optimal settings

if stream:

return self.generate\_stream(

prompt=prompt,

model\_name=model\_name,

\*\*optimal\_config

)

else:

return self.generate(

prompt=prompt,

model\_name=model\_name,

options=GenerationOptions(\*\*optimal\_config)

)

async def auto\_select\_model(self, prompt: str, task\_type: str) -> str:

"""Enhanced model selection considering task type"""

# Task-specific model preferences

task\_models = {

"code\_generation": "deepcoder:14b",

"technical\_analysis": "deepseek-r1:32b",

"creative\_writing": "deepseek-r1:latest",

"conversational": "mistral:latest",

"summarization": "deepseek-r1:latest"

}

preferred\_model = task\_models.get(task\_type, "deepseek-r1:latest")

# Verify model availability and resources

if await self.check\_resources\_for\_model(preferred\_model):

return preferred\_model

# Fallback chain

fallback\_chain = [

"deepseek-r1:latest",

"mistral:latest",

"deepcoder:14b"

]

for model in fallback\_chain:

if await self.check\_resources\_for\_model(model):

return model

return "mistral:latest" # Final fallback

**3. Enhanced AG-UI Integration**

**3.1 CopilotKit Actions with Advanced Configuration**

typescript

// Enhanced CopilotKit actions with configuration management

import { useCopilotAction } from '@copilotkit/react-core';

import { AGSlider, AGSelect, AGSwitch } from '@ag-ui/components';

export const useAdvancedGenerationAction = () => {

return useCopilotAction({

name: "generate\_with\_config",

description: "Generate text with advanced configuration options",

parameters: [

{ name: "prompt", type: "string", description: "Text prompt" },

{ name: "task\_type", type: "string", description: "Type of task",

enum: ["creative\_writing", "technical\_analysis", "code\_generation", "conversational", "summarization"] },

{ name: "temperature", type: "number", description: "Creativity level (0.1-1.0)" },

{ name: "model\_preference", type: "string", description: "Preferred model" },

{ name: "stream", type: "boolean", description: "Enable streaming" }

],

handler: async (params) => {

const response = await fetch('/api/generate/enhanced', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({

prompt: params.prompt,

task\_type: params.task\_type,

model: params.model\_preference,

stream: params.stream,

user\_preferences: {

temperature: params.temperature

}

})

});

if (params.stream) {

// Handle streaming response

const reader = response.body?.getReader();

let result = "";

while (true) {

const { done, value } = await reader!.read();

if (done) break;

const chunk = new TextDecoder().decode(value);

const lines = chunk.split('\n');

for (const line of lines) {

if (line.startsWith('data: ')) {

try {

const data = JSON.parse(line.slice(6));

if (data.content) {

result += data.content;

}

} catch (e) {

// Skip malformed JSON

}

}

}

}

return result;

} else {

const data = await response.json();

return data.response;

}

}

});

};

// Advanced configuration UI component

export const AdvancedConfigPanel: React.FC = () => {

const [taskType, setTaskType] = useState('conversational');

const [temperature, setTemperature] = useState(0.7);

const [modelPreference, setModelPreference] = useState('auto');

const [streamingEnabled, setStreamingEnabled] = useState(true);

const generateAction = useAdvancedGenerationAction();

return (

<AGCard>

<AGCard.Header>

<AGTypography variant="h6">Advanced Generation Settings</AGTypography>

</AGCard.Header>

<AGCard.Content>

<AGGrid container spacing={2}>

<AGGrid item xs={12}>

<AGSelect

label="Task Type"

value={taskType}

onChange={(e) => setTaskType(e.target.value)}

options={[

{ value: 'conversational', label: 'Conversational' },

{ value: 'technical\_analysis', label: 'Technical Analysis' },

{ value: 'code\_generation', label: 'Code Generation' },

{ value: 'creative\_writing', label: 'Creative Writing' },

{ value: 'summarization', label: 'Summarization' }

]}

/>

</AGGrid>

<AGGrid item xs={12}>

<AGTypography variant="body2">

Creativity Level: {temperature}

</AGTypography>

<AGSlider

value={temperature}

onChange={(value) => setTemperature(value)}

min={0.1}

max={1.0}

step={0.1}

marks={[

{ value: 0.1, label: 'Precise' },

{ value: 0.5, label: 'Balanced' },

{ value: 1.0, label: 'Creative' }

]}

/>

</AGGrid>

<AGGrid item xs={12}>

<AGSelect

label="Model Preference"

value={modelPreference}

onChange={(e) => setModelPreference(e.target.value)}

options={[

{ value: 'auto', label: 'Auto-select' },

{ value: 'mistral:latest', label: 'Mistral (Fast)' },

{ value: 'deepseek-r1:latest', label: 'DeepSeek (Balanced)' },

{ value: 'deepcoder:14b', label: 'DeepCoder (Code)' },

{ value: 'deepseek-r1:32b', label: 'DeepSeek 32B (Premium)' }

]}

/>

</AGGrid>

<AGGrid item xs={12}>

<AGSwitch

label="Enable Streaming"

checked={streamingEnabled}

onChange={(checked) => setStreamingEnabled(checked)}

/>

</AGGrid>

</AGGrid>

</AGCard.Content>

</AGCard>

);

};

**4. Memory Management Enhancement**

**4.1 Multi-Turn Conversation Support**

python

# Enhanced memory management for AG-UI conversations

class ConversationMemoryManager:

"""Advanced conversation memory with context management"""

def \_\_init\_\_(self, max\_context\_length: int = 8192):

self.conversations = {} # user\_id -> conversation\_data

self.max\_context\_length = max\_context\_length

def create\_conversation(

self,

user\_id: str,

conversation\_id: str = None

) -> str:

"""Create new conversation session"""

conv\_id = conversation\_id or f"{user\_id}\_{int(time.time())}"

self.conversations[conv\_id] = {

"user\_id": user\_id,

"messages": [],

"context\_length": 0,

"created\_at": datetime.utcnow(),

"last\_activity": datetime.utcnow(),

"metadata": {

"model\_used": [],

"total\_tokens": 0,

"conversation\_type": "general"

}

}

return conv\_id

def add\_message(

self,

conversation\_id: str,

role: str,

content: str,

metadata: Dict[str, Any] = None

):

"""Add message to conversation with context management"""

if conversation\_id not in self.conversations:

raise ValueError(f"Conversation {conversation\_id} not found")

conv = self.conversations[conversation\_id]

message = {

"role": role,

"content": content,

"timestamp": datetime.utcnow().isoformat(),

"metadata": metadata or {}

}

conv["messages"].append(message)

conv["context\_length"] += len(content)

conv["last\_activity"] = datetime.utcnow()

# Trim context if too long

if conv["context\_length"] > self.max\_context\_length:

self.\_trim\_context(conversation\_id)

def \_trim\_context(self, conversation\_id: str):

"""Intelligent context trimming preserving important messages"""

conv = self.conversations[conversation\_id]

# Always preserve first message (system/initial context)

# and last few exchanges

messages\_to\_keep = []

current\_length = 0

# Keep system message if present

if conv["messages"] and conv["messages"][0]["role"] == "system":

messages\_to\_keep.append(conv["messages"][0])

current\_length += len(conv["messages"][0]["content"])

# Keep recent messages (work backwards)

for message in reversed(conv["messages"][-10:]): # Last 10 messages

message\_length = len(message["content"])

if current\_length + message\_length <= self.max\_context\_length \* 0.8:

messages\_to\_keep.insert(-1 if messages\_to\_keep else 0, message)

current\_length += message\_length

else:

break

conv["messages"] = messages\_to\_keep

conv["context\_length"] = current\_length

def get\_conversation\_context(

self,

conversation\_id: str,

format\_for\_llm: bool = True

) -> Union[List[Dict], str]:

"""Get conversation context formatted for LLM"""

if conversation\_id not in self.conversations:

return [] if format\_for\_llm else ""

messages = self.conversations[conversation\_id]["messages"]

if format\_for\_llm:

return messages

else:

# Format as string for simple clients

formatted = []

for msg in messages:

role = msg["role"].capitalize()

content = msg["content"]

formatted.append(f"{role}: {content}")

return "\n".join(formatted)

# Integration with FastAPI

class ConversationAwareOllamaGateway(CitadelOllamaGateway):

"""Ollama gateway with conversation memory"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.memory\_manager = ConversationMemoryManager()

async def generate\_with\_conversation(

self,

prompt: str,

conversation\_id: str,

user\_id: str,

model: str = "deepseek-r1:latest",

\*\*generation\_config

):

"""Generate response with conversation context"""

# Create conversation if it doesn't exist

if conversation\_id not in self.memory\_manager.conversations:

self.memory\_manager.create\_conversation(user\_id, conversation\_id)

# Add user message to conversation

self.memory\_manager.add\_message(

conversation\_id,

"user",

prompt,

{"model\_requested": model}

)

# Get conversation context

context\_messages = self.memory\_manager.get\_conversation\_context(

conversation\_id,

format\_for\_llm=True

)

# Generate response using context

client = self.get\_client(model)

# Format messages for Ollama (simple prompt format)

full\_context = self.memory\_manager.get\_conversation\_context(

conversation\_id,

format\_for\_llm=False

)

response = await client.generate(

prompt=full\_context + f"\nUser: {prompt}\nAssistant:",

\*\*generation\_config

)

# Add assistant response to conversation

self.memory\_manager.add\_message(

conversation\_id,

"assistant",

response,

{"model\_used": model, \*\*generation\_config}

)

return {

"response": response,

"conversation\_id": conversation\_id,

"context\_length": self.memory\_manager.conversations[conversation\_id]["context\_length"],

"message\_count": len(self.memory\_manager.conversations[conversation\_id]["messages"])

}

# Enhanced FastAPI endpoints

@app.post("/api/chat/conversation")

async def conversation\_chat(

request: ConversationChatRequest,

gateway: ConversationAwareOllamaGateway = Depends(get\_gateway)

):

"""Chat with conversation memory"""

return await gateway.generate\_with\_conversation(

prompt=request.prompt,

conversation\_id=request.conversation\_id,

user\_id=request.user\_id,

model=request.model,

temperature=request.temperature,

top\_p=request.top\_p,

repeat\_penalty=request.repeat\_penalty

)

@app.get("/api/conversations/{conversation\_id}")

async def get\_conversation(

conversation\_id: str,

gateway: ConversationAwareOllamaGateway = Depends(get\_gateway)

):

"""Get conversation history"""

if conversation\_id not in gateway.memory\_manager.conversations:

raise HTTPException(status\_code=404, detail="