Cheat Sheet: BeeAI & AG2 (AutoGen) Frameworks for Building Agentic AI Systems

Estimated time: 15 minutes

BeeAI framework - Production-ready AI agents

What is BeeAI?

BeeAI is an open-source platform for building production-ready AI agents, developed under the Linux Foundation AI & Data program.

Key Features:

- **Production-ready**: Built-in caching, monitoring, and OpenTelemetry integration
- **Provider-agnostic**: Supports 10+ LLM providers (OpenAI, watsonx.ai, Groq, Ollama)
- **Advanced patterns**: ReAct reasoning, systematic thinking, and multi-agent coordination

---

BeeAI core usage

Key library imports

```
import asyncio
from beeai_framework.backend import ChatModel, ChatModelParameters, SystemMessage, UserMessage
from beeai_framework.agents.experimental import RequirementAgent
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.tools.search.wikipedia import WikipediaTool
from beeai_framework.agents.experimental.requirements.conditional import ConditionalRequirement
from beeai_framework.tools.think import ThinkTool
from beeai_framework.agents.experimental.requirements.ask_permission import AskPermissionRequirement
from beeai_framework.tools.handoff import HandoffTool
from beeai_framework.tools import Tool
```

Basic usage

```
# Initialize model
llm = ChatModel.from_name("watsonx:ibm/granite-3-3-8b-instruct", ChatModelParameters(temperature=0))
# Define conversation
messages = [
    SystemMessage(content="You are a helpful AI assistant."),
    UserMessage(content="Explain machine learning in simple terms.")
]
# Run asynchronously
async def main():
    response = await llm.create(messages=messages)
    print(response.get_text_content())
asyncio.run(main())
```

Structured outputs

```
class BusinessPlan(BaseModel):
    business_name: str = Field(description="Catchy name for the business")
    elevator_pitch: str = Field(description="30-second description")
    revenue_streams: List[str] = Field(description="Ways to make money")
# Generate structured output
async def main():
    llm = ChatModel.from_name("openai:gpt-5-nano", ChatModelParameters(temperature=0))
    response = await llm.create_structure(
        schema=BusinessPlan,
        messages=[
            SystemMessage("You are a business consultant."),
            UserMessage("Create a business plan for a food delivery app.")
        ]
    )
```

```
    print(response.object)  # Returns typed BusinessPlan object
asyncio.run(main())
```

BeeAI agents

Basic agent setup

```
# Initialize model
llm = ChatModel.from_name("watsonx:ibm/granite-3-3-8b-instruct", ChatModelParameters(temperature=0))
# Basic agent
agent = RequirementAgent(
    llm=llm,
    memory=UnconstrainedMemory(),
    instructions="You are an AI assistant specialized in data analysis."
)
# Run agent
async def main():
    result = await agent.run("What is machine learning?")
    print(f"Answer: {result.answer.text}")
asyncio.run(main())
```

Adding tools (common pattern)

```
# Initialize model
llm = ChatModel.from_name("watsonx:ibm/granite-3-3-8b-instruct", ChatModelParameters(temperature=0))
# Agent with tool
agent = RequirementAgent(
    llm=llm,
    memory=UnconstrainedMemory(),
    instructions="You are a research assistant.",
    tools=[WikipediaTool()],
    requirements=[ConditionalRequirement(WikipediaTool, max_invocations=1)]
)
# Run agent
async def main():
    result = await agent.run("What is machine learning?")
    print(f"Answer: {result.answer.text}")
asyncio.run(main())
```

ReAct agent

```
# Initialize model
llm = ChatModel.from_name("watsonx:ibm/granite-3-3-8b-instruct", ChatModelParameters(temperature=0))
# ReAct agent with Think → Act → Think → Act cycle
agent = RequirementAgent(
    llm=llm,
    memory=UnconstrainedMemory(),
    instructions="You are a helpful assistant.",
    tools=[ThinkTool(), WikipediaTool()],
    requirements=[ConditionalRequirement(
        ThinkTool,
        force_at_step=1,           # Think first
        force_after=Tool,          # Think after every tool use
        consecutive_allowed=False, # No consecutive thinking
        max_invocations=3         # Limit thinking cycles
    )]
)
# Run agent
async def main():
    result = await agent.run("What is machine learning?")
```

```
    print(f"Answer: {result.answer.text}")
asyncio.run(main())
```

Human-in-the-loop

```
# Initialize model
llm = ChatModel.from_name("watsonx:ibm/granite-3-3-8b-instruct", ChatModelParameters(temperature=0))
# Secure agent requiring approval
agent = RequirementAgent(
    llm=llm,
    memory=UnconstrainedMemory(),
    instructions="You are a helpful assistant.",
    tools=[WikipediaTool(), ThinkTool()],
    requirements=[
        AskPermissionRequirement(WikipediaTool),  # Ask before using Wikipedia
        ConditionalRequirement(ThinkTool, force_at_step=1, max_invocations=3)
    ]
)
# Run agent
async def main():
    result = await agent.run("What is machine learning?")
    print(f"Answer: {result.answer.text}")
asyncio.run(main())
```

Multi-agent handoffs (code mockup, replace all instances of . . . to get a working example)

```
# Initialize model
llm = ChatModel.from_name("watsonx:ibm/granite-3-3-8b-instruct", ChatModelParameters(temperature=0))
# Create specialized agents
specialist_agent1 = RequirementAgent(...)
specialist_agent2 = RequirementAgent(...)
# Create handoff tools
handoff_to_agent1 = HandoffTool(
    specialist_agent1,
    name="DataAnalyst",
    description="Consult the data analysis specialist"
)
handoff_to_agent2 = HandoffTool(
    specialist_agent2,
    name="ReportWriter",
    description="Consult the report writing specialist"
)
# Coordinator agent
coordinator = RequirementAgent(
    llm=llm,
    memory=UnconstrainedMemory(),
    instructions="You coordinate tasks between specialists.",
    tools=[handoff_to_agent1, handoff_to_agent2, ThinkTool()]
)
# Run coordinator agent
async def main():
    result = await coordinator.run(...)
    print(f"Answer: {result.answer.text}")
asyncio.run(main())
```

AG2 framework: Multi-agent workflows

What is AG2?

AG2 (formerly AutoGen) is an open-source framework for multi-agent AI collaboration through structured interactions.

Core strengths:

- **Simple multi-agent setup**: Easy agent collaboration
- **Human integration**: Seamless oversight and control
- **Proven patterns**: Battle-tested orchestration methods

AG2 setup

```
pip install ag2[openai]
from autogen import ConversableAgent, AssistantAgent, UserProxyAgent
from autogen import GroupChat, GroupChatManager
from autogen.llm_config import LLMConfig
llm_config = LLMConfig(api_type="openai", model="gpt-4o-mini")
```

AG2 core patterns

Two-agent conversation (simplest pattern)

```
# Create specialized agents
with llm_config:
    student = ConversableAgent(
        name="student",
        system_message="You are a curious student who asks clear questions",
        human_input_mode="NEVER"
    )

    tutor = ConversableAgent(
        name="tutor",
        system_message="You are a helpful tutor with clear explanations",
        human_input_mode="NEVER"
    )
# Start conversation
chat_result = student.initiate_chat(
    recipient=tutor,
    message="Can you explain what a neural network is?",
    max_turns=2,
    summary_method="reflection_with_llm"
)
print("Final Summary:")
print(chat_result.summary)
```

Code generation & execution

```
# Code generation and execution example
assistant = AssistantAgent(
    name="assistant",
    system_message="Helpful assistant who writes clear Python code"
)
user_proxy = UserProxyAgent(
    name="user_proxy",
    human_input_mode="NEVER",
    max_consecutive_auto_reply=5,
    code_execution_config={
        "executor": LocalCommandLineCodeExecutor(work_dir="coding")
    }
)
# Task execution
user_proxy.initiate_chat(
    recipient=assistant,
    message="Plot a sine wave using matplotlib and save as sine_wave.png"
)
```

Group chat (multiple agents)

```
# Create specialized education agents
lesson_planner = ConversableAgent(
    name="planner_agent",
    system_message="Create lesson plans for 4th graders",
    description="Makes lesson plans"
)
lesson_reviewer = ConversableAgent(
    name="reviewer_agent",
    system_message="Review plans and suggest up to 3 brief edits",
    description="Reviews lesson plans and suggests edits"
)
teacher = ConversableAgent(
    name="teacher_agent",
    system_message="Suggest topics and reply DONE when satisfied",
    is_termination_msg=lambda x: "DONE" in x.get("content", "").upper()
)
# Configure group chat
groupchat = GroupChat(
    agents=[teacher, lesson_planner, lesson_reviewer],
    speaker_selection_method="auto"
)
manager = GroupChatManager(
    name="group_manager",
    groupchat=groupchat,
    llm_config=llm_config
)
# Start collaborative workflow
teacher.initiate_chat(
    recipient=manager,
    message="Make a simple lesson about the moon.",
    max_turns=6,
    summary_method="reflection_with_llm"
)
```

AG2 human oversight

Human input modes

- **"ALWAYS"**: Human approves every response
- **"NEVER"**: Fully autonomous
- **"TERMINATE"**: The human decides when to end

```
# Bug triage system with human oversight
triage_bot = ConversableAgent(
    name="triage_bot",
    system_message="""You are a bug triage assistant. For each bug report:
    - Urgent issues (crash, security, data loss): escalate and ask for confirmation
    - Minor issues (cosmetic, typos): suggest closing but ask for review
    - Otherwise: classify as medium priority and ask for review""",
    llm_config=llm_config
)
human = ConversableAgent(
    name="human",
    human_input_mode="ALWAYS"  # Human reviews each decision
)
```

AG2 tools & structured output

Custom tools

```
def is_prime(n: int) -> str:
    """Check if a number is prime"""
```

```
        if n < 2: return "No"
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0: return "No"
        return "Yes"
    register_function(
        is_prime,
        caller=math_asker,      # Agent that requests the tool
        executor=math_checker,  # Agent that executes the tool
        description="Check if a number is prime. Returns Yes or No."
    )
```

Structured outputs

```
from pydantic import BaseModel
class TicketSummary(BaseModel):
    customer_name: str
    issue_type: str
    urgency_level: str
    recommended_action: str
llm_config = LLMConfig(
    api_type="openai",
    model="gpt-4o-mini",
    response_format=TicketSummary  # Enforces structure
)
```

Quick decision guide

Need	Use BeeAI when	Use AG2 when
Production deployment	Need enterprise features, monitoring	Simple, proven patterns sufficient
Human oversight	Complex approval workflows	Basic human-in-the-loop needed
Multi-agent coordination	Need fine-grained control	Want simple group collaboration
Tool integration	Custom tools with requirements	Basic function registration
Getting started	Have specific production needs	Want to prototype quickly

---

Essential best practices

Security

- **Never hardcode API keys**: Use environment variables
- Set **max_consecutive_auto_reply** to prevent infinite loops
- Use **human oversight** for high-stakes decisions

Agent design

- Write **clear system messages** defining role and constraints
- Use **specialized agents** for specific tasks rather than generalists
- Set **termination conditions** to end conversations cleanly

Production tips

- Test with **low max_turns** first to avoid token costs
- Use **temperature=0** for consistent outputs
- Monitor **conversation quality** and intervene when needed

BeeAI: Production-ready, enterprise features, fine-grained control
AG2: Simple, proven, great for prototyping and education

Author(s)

Wojciech "Victor" Fulmyk