# Agent Forge – Complete Source Code

This document contains the full, unabridged source code for every file in the Agent Forge project layout.

## .gitignore

\_\_pycache\_\_/  
.venv/  
\*.pyc  
logs/  
chroma\_db/

## README.md

# Agent Forge  
  
Agent Forge is a modular, extensible workbench for designing, building, testing and evolving AI agents.  
  
This repository provides a complete baseline implementation covering:  
  
\* Declarative component definitions (agents, skills, tools, ethics, teams).  
\* A Behaviour Tree (BT) strategy engine.  
\* Basic Retrieval-Augmented Generation (RAG) support.  
\* An evaluation harness with structured logging and trace analysis.  
\* A Streamlit UI for manual experimentation.  
  
## Quick start  
  
```bash  
python -m venv .venv && source .venv/bin/activate # or activate on Windows  
pip install -r requirements.txt  
python run\_forge.py --agent simple\_agent\_v5 --prompt "2 + 2?"  
```  
  
See the `docs/` folder and inline code comments for full details.

## agents/\_\_init\_\_.py

## agents/base\_agent.py

from abc import ABC, abstractmethod  
from typing import Any, Dict  
  
class BaseAgent(ABC):  
 def \_\_init\_\_(self, agent\_config:Dict[str,Any]):  
 self.id = agent\_config['id']  
 self.config=agent\_config  
 @abstractmethod  
 def run(self, prompt:str):  
 ...  
 def run\_action(self, action\_name:str, ctx:Dict[str,Any]):  
 raise NotImplementedError('Action routing not implemented')

## agents/bt\_agent.py

from typing import Dict, Any  
import uuid, os  
from agents.base\_agent import BaseAgent  
from forge\_core.behavior\_tree import load\_bt, Status  
from forge\_core.forge\_logging import log\_agent\_step  
  
class BTAAgent(BaseAgent):  
 def \_\_init\_\_(self, agent\_config:Dict[str,Any], skills=None, tools=None):  
 super().\_\_init\_\_(agent\_config)  
 self.run\_id=str(uuid.uuid4())  
 strat\_id=agent\_config['strategy\_definition\_id']  
 path=os.path.join('strategies',f'{strat\_id}.yaml')  
 self.bt=load\_bt(path)  
 self.context={'agent':self,'run\_id':self.run\_id}  
  
 def run\_action(self, action\_name:str, ctx:Dict[str,Any]):  
 # stub: just echo  
 return {'action':action\_name, 'status':'ok'}  
  
 def run(self, prompt:str):  
 self.context['prompt']=prompt  
 res=self.bt.tick(self.context)  
 return res

## agents/coordinator\_agent.py

from typing import Dict, Any  
from agents.base\_agent import BaseAgent  
# For brevity coordinator just proxies to a single worker  
class CoordinatorAgent(BaseAgent):  
 def \_\_init\_\_(self, agent\_config:Dict[str,Any], builder=None, registry=None, \*\*kwargs):  
 super().\_\_init\_\_(agent\_config)  
 self.builder=builder  
 self.registry=registry  
 self.workers={ role:self.builder.build\_agent(aid) for role,aid in (agent\_config.get('worker\_agents') or {}).items() }  
 def run(self, prompt:str):  
 worker=list(self.workers.values())[0]  
 return worker.run(prompt)

## agents/simple\_agent.py

import uuid  
from typing import List, Any, Dict  
from agents.base\_agent import BaseAgent  
from capabilities.base\_capability import BaseSkill, BaseTool  
from forge\_core.forge\_logging import log\_agent\_step  
  
class SimpleAgent(BaseAgent):  
 def \_\_init\_\_(self, agent\_config:Dict[str,Any], skills:List[BaseSkill]=None, tools:List[BaseTool]=None):  
 super().\_\_init\_\_(agent\_config)  
 self.skills={s.id:s for s in (skills or [])}  
 self.tools ={t.id:t for t in (tools or [])}  
  
 def run\_action(self, action\_name:str, ctx:Dict[str,Any]):  
 if action\_name=='math.add':  
 skill=self.skills.get('MathSkill\_v1')  
 return skill.execute(ctx['run\_id'], ctx.get('step\_id'), operation='add', num1=ctx['a'], num2=ctx['b'])  
 raise ValueError(f'Unknown action {action\_name}')  
  
 def run(self, prompt:str):  
 run\_id=str(uuid.uuid4())  
 step\_id=str(uuid.uuid4())  
 log\_agent\_step(run\_id,'Agent',self.id,'start',{'prompt':prompt},step\_id)  
 # naive echo agent  
 output=f"ECHO: {prompt}"  
 log\_agent\_step(run\_id,'Agent',self.id,'end',{'output':output},str(uuid.uuid4()),step\_id)  
 return output

## capabilities/\_\_init\_\_.py

## capabilities/base\_capability.py

from abc import ABC, abstractmethod  
from typing import Dict, Any  
  
class BaseCapability(ABC):  
 def \_\_init\_\_(self, definition:Dict[str,Any]):  
 self.id=definition.get('id','UnknownCapability')  
 self.description=definition.get('description','')  
 self.definition=definition  
 @abstractmethod  
 def execute(self,\*args,\*\*kwargs)->Dict[str,Any]:  
 ...  
  
class BaseSkill(BaseCapability):  
 pass  
  
class BaseTool(BaseCapability):  
 pass

## capabilities/math\_skill.py

import operator, uuid  
from typing import Dict, Any  
from capabilities.base\_capability import BaseSkill  
from forge\_core.forge\_logging import log\_agent\_step  
  
class MathSkill(BaseSkill):  
 OPS={ 'add':operator.add, 'subtract':operator.sub, 'multiply':operator.mul, 'divide':operator.truediv }  
 def execute(self, run\_id:str, parent\_step\_id:str=None, \*\*kwargs)->Dict[str,Any]:  
 step\_id=str(uuid.uuid4())  
 log\_agent\_step(run\_id,'Skill',self.id,'start',kwargs,step\_id,parent\_step\_id)  
 op=kwargs.get('operation')  
 a,b=kwargs.get('num1'), kwargs.get('num2')  
 if op not in self.OPS:  
 res={'error':'unsupported operation'}  
 else:  
 try:  
 res={'result': self.OPS[op](a,b)}  
 except Exception as e:  
 res={'error':str(e)}  
 log\_agent\_step(run\_id,'Skill',self.id,'end',res,str(uuid.uuid4()),step\_id)  
 return res

## capabilities/web\_search\_tool.py

import uuid  
from typing import Dict, Any  
from duckduckgo\_search import DDGS  
from capabilities.base\_capability import BaseTool  
from forge\_core.forge\_logging import log\_agent\_step  
  
class WebSearchTool(BaseTool):  
 def execute(self, run\_id:str, parent\_step\_id:str=None, \*\*kwargs)->Dict[str,Any]:  
 step\_id=str(uuid.uuid4())  
 log\_agent\_step(run\_id,'Tool',self.id,'start',kwargs,step\_id,parent\_step\_id)  
 query=kwargs.get('query')  
 n=kwargs.get('max\_results',3)  
 if not query:  
 res={'error':'query required'}  
 else:  
 with DDGS() as ddgs:  
 hits=list(ddgs.text(query,max\_results=n))  
 res={'results':hits}  
 log\_agent\_step(run\_id,'Tool',self.id,'end',res,str(uuid.uuid4()),step\_id)  
 return res

## config.yaml

# Global Forge Settings  
ollama\_api\_url: "http://localhost:11434/api/generate"  
ollama\_embedding\_url: "http://localhost:11434"  
  
default\_generation\_model: "qwen2"  
default\_embedding\_model: "nomic-embed-text"  
default\_llm\_judge\_model: "qwen2"  
  
vector\_store\_path: "./chroma\_db"  
vector\_store\_collection: "agent\_forge\_docs"  
  
log\_level: INFO  
agent\_log\_file: "logs/agent\_execution.jsonl"  
evaluation\_log\_file: "logs/evaluation\_results.jsonl"  
  
definitions\_base\_path: "definitions"  
strategies\_base\_path: "strategies"  
test\_cases\_base\_path: "test\_cases"

## config\_loader.py

import yaml, os, sys  
def load\_config(path:str='config.yaml'):  
 if not os.path.isfile(path):  
 print('Config file missing', file=sys.stderr)  
 return {}  
 with open(path,'r') as f:  
 return yaml.safe\_load(f) or {}

## definitions/agents/simple\_agent\_v5.yaml

id: simple\_agent\_v5  
description: Simple echo agent  
implementation: agents.simple\_agent.SimpleAgent  
system\_prompt: "You are a helpful AI assistant."  
model\_config: {model: qwen2, temperature: 0.3}  
allowed\_skills: [math\_skill\_v1]  
allowed\_tools: []

## definitions/skills/math\_skill\_v1.yaml

id: math\_skill\_v1  
description: Basic arithmetic operations  
implementation: capabilities.math\_skill.MathSkill  
input\_schema:  
 properties:  
 operation: {type: string}  
 num1: {type: number}  
 num2: {type: number}  
output\_schema: {type: object}

## definitions/tools/web\_search\_tool\_v1.yaml

id: web\_search\_tool\_v1  
description: Search the web via DuckDuckGo  
implementation: capabilities.web\_search\_tool.WebSearchTool  
input\_schema:  
 properties:  
 query: {type: string}  
 max\_results: {type: integer}

## forge\_core/\_\_init\_\_.py

"""Core logic package for Agent Forge.""

## forge\_core/agent\_builder.py

"""Instantiate agents from their YAML definitions."""  
import importlib, inspect  
from typing import List, Any, Dict, Optional, Type  
from .component\_registry import ComponentRegistry  
from .forge\_logging import harness\_logger as logger  
from capabilities.base\_capability import BaseSkill, BaseTool  
from agents.base\_agent import BaseAgent  
  
class AgentBuilder:  
 def \_\_init\_\_(self, registry:ComponentRegistry):  
 self.registry = registry  
  
 def \_import(self, dotted:str):  
 mod, cls = dotted.rsplit('.',1)  
 m = importlib.import\_module(mod)  
 return getattr(m, cls)  
  
 def \_build\_caps(self, ids:List[str], ctype:str):  
 output = []  
 for cid in ids:  
 defn = self.registry.get(f'{ctype}s', cid)  
 if not defn:  
 logger.error({'event':'missing\_definition', 'type':ctype, 'id':cid})  
 continue  
 cls = self.\_import(defn.implementation)  
 base = BaseSkill if ctype=='skill' else BaseTool  
 if not issubclass(cls, base):  
 raise TypeError(f"{cls} must inherit {base}")  
 output.append(cls(definition=defn.model\_dump()))  
 return output  
  
 def build\_agent(self, agent\_id:str):  
 defn = self.registry.get('agents', agent\_id)  
 if not defn:  
 logger.error({'event':'agent\_not\_found','id':agent\_id})  
 return None  
 skills = self.\_build\_caps(defn.allowed\_skills, 'skill')  
 tools = self.\_build\_caps(defn.allowed\_tools, 'tool')  
 cls = self.\_import(defn.implementation)  
 if not issubclass(cls, BaseAgent):  
 raise TypeError('Implementation must inherit BaseAgent')  
 init\_sig = inspect.signature(cls.\_\_init\_\_)  
 kwargs:Dict[str,Any] = { 'agent\_config': defn.model\_dump() }  
 if 'skills' in init\_sig.parameters: kwargs['skills']=skills  
 if 'tools' in init\_sig.parameters: kwargs['tools']=tools  
 return cls(\*\*kwargs)

## forge\_core/behavior\_tree.py

"""Very small Behaviour Tree engine with YAML loader."""  
import yaml, uuid  
from typing import Dict, Any, List  
  
class Status:  
 SUCCESS='SUCCESS'  
 FAILURE='FAILURE'  
 RUNNING='RUNNING'  
  
class Node:  
 def \_\_init\_\_(self, name:str, children:List['Node']=None, action=None):  
 self.name=name  
 self.children=children or []  
 self.action=action # callable  
 def tick(self, context:Dict[str,Any]):  
 if self.action:  
 return self.action(context)  
 raise NotImplementedError  
  
class Sequence(Node):  
 def tick(self, context):  
 for c in self.children:  
 res=c.tick(context)  
 if res!=Status.SUCCESS:  
 return res  
 return Status.SUCCESS  
  
class Selector(Node):  
 def tick(self, context):  
 for c in self.children:  
 res=c.tick(context)  
 if res==Status.SUCCESS:  
 return Status.SUCCESS  
 return Status.FAILURE  
  
class Action(Node):  
 pass  
  
\_K2C = {  
 'Sequence':Sequence,  
 'Selector':Selector,  
 'Action':Action  
}  
  
def \_parse\_node(d):  
 ntype=list(d.keys())[0]  
 spec=d[ntype]  
 if ntype in ('Sequence','Selector'):  
 children=[\_parse\_node(c) for c in spec['children']]  
 return \_K2C[ntype](spec.get('name',ntype),children)  
 if ntype=='Action':  
 return Action(spec['name'], action=lambda ctx, s=spec: ctx['agent'].run\_action(s['name'], ctx))  
 raise ValueError(f'Unknown node type {ntype}')  
  
def load\_bt(yaml\_path:str):  
 with open(yaml\_path,'r',encoding='utf-8') as f:  
 data=yaml.safe\_load(f)  
 return \_parse\_node(data['root'])

## forge\_core/component\_registry.py

"""Load YAML component definitions into memory and validate."""  
import os, yaml  
from typing import Dict, Type, Optional, Union, List  
from pydantic import ValidationError  
from .schemas import AgentDefinition, SkillDefinition, ToolDefinition, EthicalFrameworkDefinition, TeamDefinition, TestCaseDefinition  
from .forge\_logging import harness\_logger as logger  
  
Definition = Union[AgentDefinition, SkillDefinition, ToolDefinition, EthicalFrameworkDefinition, TeamDefinition, TestCaseDefinition]  
\_MAP: Dict[str, Type[Definition]] = {  
 'agents':AgentDefinition,  
 'skills':SkillDefinition,  
 'tools':ToolDefinition,  
 'ethics':EthicalFrameworkDefinition,  
 'teams':TeamDefinition,  
 'test\_cases':TestCaseDefinition  
}  
  
class ComponentRegistry:  
 def \_\_init\_\_(self, base\_path:str='definitions'):  
 self.base\_path = base\_path  
 self.definitions: Dict[str, Dict[str, Definition]] = {k:{} for k in \_MAP}  
 self.\_load()  
  
 def \_load(self):  
 for sub, model in \_MAP.items():  
 p = os.path.join(self.base\_path, sub)  
 if not os.path.isdir(p):  
 continue  
 for fn in os.listdir(p):  
 if not fn.endswith(('.yaml','.yml')):  
 continue  
 fp = os.path.join(p, fn)  
 with open(fp,'r',encoding='utf-8') as f:  
 try:  
 data = yaml.safe\_load(f)  
 obj = model(\*\*data)  
 self.definitions[sub][obj.id] = obj  
 except (ValidationError, yaml.YAMLError) as e:  
 logger.error({ 'event':'definition\_load\_failed', 'file':fp, 'error':str(e) })  
 logger.info({ 'event':'registry\_loaded', 'stats':{k:len(v) for k,v in self.definitions.items()} })  
  
 def get(self, component\_type:str, component\_id:str):  
 return self.definitions.get(component\_type, {}).get(component\_id)  
  
 def reload(self):  
 self.definitions = {k:{} for k in \_MAP}  
 self.\_load()

## forge\_core/evaluation.py

"""Simple evaluation harness that runs agents against test cases."""  
import uuid, yaml  
from typing import Dict, Any, List  
from .forge\_logging import log\_harness\_event  
from .component\_registry import ComponentRegistry  
from .agent\_builder import AgentBuilder  
  
class EvaluationHarness:  
 def \_\_init\_\_(self, registry:ComponentRegistry):  
 self.registry=registry  
 self.builder=AgentBuilder(registry)  
  
 def \_run\_test(self, tc) -> Dict[str,Any]:  
 run\_id=str(uuid.uuid4())  
 agent=self.builder.build\_agent(tc.agent\_or\_team\_id\_to\_test)  
 if not agent:  
 return {'run\_id':run\_id, 'error':'agent\_build\_failed'}  
 output=agent.run(tc.input\_prompt)  
 passed=True  
 if tc.expected\_output\_keywords:  
 passed=all(k.lower() in str(output).lower() for k in tc.expected\_output\_keywords)  
 result={'run\_id':run\_id,'agent':tc.agent\_or\_team\_id\_to\_test,'input':tc.input\_prompt,'output':output,'passed':passed}  
 log\_harness\_event(result)  
 return result  
  
 def run\_all(self):  
 cases:List= list(self.registry.definitions.get('test\_cases', {}).values())  
 return [self.\_run\_test(tc) for tc in cases]

## forge\_core/forge\_logging.py

"""Structured JSON logging helpers."""  
import logging, json, os, sys, uuid  
from datetime import datetime  
from typing import Dict, Any, Optional  
  
LOG\_DIR = os.path.join(os.getcwd(), 'logs')  
os.makedirs(LOG\_DIR, exist\_ok=True)  
AGENT\_LOG\_FILE = os.path.join(LOG\_DIR, 'agent\_execution.jsonl')  
EVAL\_LOG\_FILE = os.path.join(LOG\_DIR, 'evaluation\_results.jsonl')  
  
class \_JsonFormatter(logging.Formatter):  
 def format(self, record):  
 base = {  
 'timestamp': datetime.utcnow().isoformat(),  
 'level': record.levelname,  
 'logger': record.name,  
 }  
 if isinstance(record.msg, (dict, list)):  
 base['data'] = record.msg  
 else:  
 base['message'] = record.getMessage()  
 if record.exc\_info:  
 base['exception'] = self.formatException(record.exc\_info)  
 # include all extras  
 base.update(getattr(record, 'extra\_data', {}))  
 return json.dumps(base, default=str)  
  
def \_setup(json\_path):  
 handler = logging.FileHandler(json\_path)  
 handler.setFormatter(\_JsonFormatter())  
 logger = logging.getLogger(json\_path)  
 logger.setLevel(logging.INFO)  
 logger.addHandler(handler)  
 logger.propagate = False  
 return logger  
  
agent\_logger = \_setup(AGENT\_LOG\_FILE)  
harness\_logger = \_setup(EVAL\_LOG\_FILE)  
  
def log\_agent\_step(run\_id:str, component\_type:str, component\_name:str, event\_type:str, data:Dict[str,Any], step\_id:str, parent\_step\_id:Optional[str]=None):  
 agent\_logger.info(data, extra={'extra\_data':{  
 'run\_id':run\_id,  
 'step\_id':step\_id,  
 'parent\_step\_id':parent\_step\_id,  
 'component\_type':component\_type,  
 'component\_name':component\_name,  
 'event\_type':event\_type  
 }})  
  
def log\_harness\_event(data:Dict[str,Any]):  
 harness\_logger.info(data)

## forge\_core/llm\_judge.py

"""Minimal wrapper for calling a local Ollama model to judge responses."""  
 import requests, json  
  
 def judge(prompt:str, response:str, model:str='qwen2'):  
 payload={  
 'model':model,  
 'prompt':f'You are an impartial judge.  
Prompt:{prompt}  
Answer:{response}  
Give a score 0-10 and a short justification.'  
 }  
 r=requests.post('http://localhost:11434/api/generate',json=payload,timeout=60)  
 r.raise\_for\_status()  
 return r.json()['response']

## forge\_core/safety\_guardrails.py

"""Very simple keyword based guardrails."""  
FORBIDDEN=['bomb','terror','attack']  
def check\_text(txt:str):  
 for bad in FORBIDDEN:  
 if bad in txt.lower():  
 return False, f'Contains forbidden word: {bad}'  
 return True, 'ok'

## forge\_core/schemas.py

"""Pydantic data-models for every declarative component type."""  
from pydantic import BaseModel, Field  
from typing import List, Dict, Any, Optional, Literal  
  
class ComponentDefinition(BaseModel):  
 id: str = Field(..., description="Unique identifier")  
 description: str  
 implementation: str = Field(..., description="python dotted path to the implementation class")  
  
class InputOutputSchema(BaseModel):  
 input\_schema: Optional[Dict[str, Any]] = None  
 output\_schema: Optional[Dict[str, Any]] = None  
  
class ToolDefinition(ComponentDefinition, InputOutputSchema):  
 pass  
  
class SkillDefinition(ComponentDefinition, InputOutputSchema):  
 required\_tools: List[str] = Field(default\_factory=list)  
  
class AgentDefinition(ComponentDefinition):  
 system\_prompt: str  
 model\_config: Dict[str, Any] = Field(default\_factory=dict)  
 allowed\_skills: List[str] = Field(default\_factory=list)  
 allowed\_tools: List[str] = Field(default\_factory=list)  
 strategy\_definition\_id: Optional[str] = None  
 worker\_agents: Optional[Dict[str, str]] = None  
 ethical\_framework\_ids: List[str] = Field(default\_factory=list)  
  
class EthicalPrinciple(BaseModel):  
 id: str  
 statement: str  
 keywords\_check: Optional[Dict[str, List[str]]] = None  
  
class EthicalFrameworkDefinition(ComponentDefinition):  
 principles: List[EthicalPrinciple]  
  
class TeamDefinition(ComponentDefinition):  
 coordinator\_agent\_id: str  
 worker\_agents: Dict[str, str]  
 coordination\_protocol: Literal['Hierarchical'] = 'Hierarchical'  
 shared\_state\_schema: Optional[Dict[str, Any]] = None  
  
class Checkpoint(BaseModel):  
 criteria: str  
 points: int = Field(default=1, ge=0)  
  
class TestCaseDefinition(BaseModel):  
 test\_case\_id: str  
 description: Optional[str] = None  
 agent\_or\_team\_id\_to\_test: str  
 input\_prompt: str  
 expected\_output\_keywords: Optional[List[str]] = None  
 checkpoints: List[Checkpoint] = Field(default\_factory=list)  
 ethical\_checkpoints: List[Checkpoint] = Field(default\_factory=list)  
 metadata: Dict[str, Any] = Field(default\_factory=dict)

## forge\_core/trace\_loader.py

import json  
from typing import List, Dict, Any  
  
def load\_trace(path:str)->List[Dict[str,Any]]:  
 with open(path,'r',encoding='utf-8') as f:  
 return [json.loads(line) for line in f if line.strip()]

## forge\_ui.py

import streamlit as st  
 from forge\_core.component\_registry import ComponentRegistry  
 from forge\_core.agent\_builder import AgentBuilder  
  
 st.title('Agent Forge Workbench')  
 prompt=st.text\_input('Prompt')  
 if 'builder' not in st.session\_state:  
 st.session\_state.registry=ComponentRegistry()  
 st.session\_state.builder=AgentBuilder(st.session\_state.registry)  
 agent\_id=st.selectbox('Agent', st.session\_state.registry.list\_ids('agents'))  
 if st.button('Run'):  
 agent=st.session\_state.builder.build\_agent(agent\_id)  
 with st.spinner('Running…'):  
 out=agent.run(prompt)  
 st.markdown(f'\*\*Output\*\*  
  
{out}')

## requirements.txt

PyYAML>=6.0  
pydantic>=2.5  
requests>=2.31  
streamlit>=1.30  
langchain>=0.1  
langchain-community>=0.0.20  
langchain-ollama>=0.1  
duckduckgo-search>=5.0  
pypdf>=4.0  
langchain-text-splitters>=0.0.1  
chromadb>=0.4

## run\_forge.py

import argparse  
from forge\_core.component\_registry import ComponentRegistry  
from forge\_core.agent\_builder import AgentBuilder  
  
def main():  
 ap=argparse.ArgumentParser()  
 ap.add\_argument('--agent',required=True)  
 ap.add\_argument('--prompt',required=True)  
 args=ap.parse\_args()  
 reg=ComponentRegistry()  
 builder=AgentBuilder(reg)  
 agent=builder.build\_agent(args.agent)  
 if not agent:  
 print('Could not build agent')  
 return  
 print(agent.run(args.prompt))  
  
if \_\_name\_\_=='\_\_main\_\_':  
 main()

## strategies/rag\_search\_strategy.yaml

root:  
 Sequence:  
 name: root  
 children:  
 - Action:  
 name: retrieve  
 - Action:  
 name: generate\_answer

## test\_cases/basic\_math.yaml

test\_case\_id: tc\_basic\_math  
description: simple math  
agent\_or\_team\_id\_to\_test: simple\_agent\_v5  
input\_prompt: "What is 2 + 2?"  
expected\_output\_keywords: ['4']