

Eugene: A Practical Python Mastery Book

From Python basics to agents, Strands, Bedrock, MCP, LangChain, Neo4j Cypher and ETL — taught entirely from the Eugene biomedical knowledge-graph codebase.

Author: generated for the Eugene team Date: 2026

How to read this book

Every concept is taught with **two views side by side**:

1. *The textbook view* — what the Python feature is, why it exists.
2. *The Eugene view* — exactly where in this repository the feature is used, and what problem it solved.

The book is structured so a developer who has only basic Python can walk through it linearly and emerge able to (a) **explain** Eugene to a teammate, (b) **operate** the agents, MCP server, and ingestion pipeline, and (c) **extend** the platform with new tools, skills, and routes.

Tip: keep `agents/eugene-agent-ws/src/query/agent/eugene_data_agent.py` open while you read Parts IV–V. Almost every advanced concept in this book appears in that one file.

PART 0 — Python from absolute zero

If you have never written Python before — start here. If you have, skim the headings; one or two will still surprise you.

Chapter 0.1. What is a Python program?

A Python program is a text file ending in `.py`. The Python interpreter reads it from top to bottom and executes one statement at a time.

```
# hello.py
print("hello, Eugene")
```

Run it with `python3 hello.py`. The interpreter prints `hello, Eugene` and exits.

That is the whole language model — *statements run in order*. Functions and classes are just statements that *define a value* (a function, a class) and bind it to a name for later use.

Chapter 0.2. Values and types — Python's eight building blocks

Every value in Python has a **type**. The eight you will see in Eugene every day:

Type	Literal	What it is
<code>int</code>	<code>42, -1</code>	whole number
<code>float</code>	<code>3.14, 60.0</code>	decimal number
<code>str</code>	<code>"hello", 'x'</code>	text — a sequence of characters
<code>bool</code>	<code>True, False</code>	yes/no
<code>None</code>	<code>None</code>	"no value" — the absence of a result
<code>list</code>	<code>[1, 2, 3]</code>	ordered, mutable sequence
<code>dict</code>	<code>{"k": 1}</code>	key→value mapping (the most important type in agents)

set	{1, 2, 3}	unordered unique collection
-----	-----------	-----------------------------

Find each one in Eugene's code:

```

_AGENT_MAX_ITERATIONS = 20                # int
_AGENT_STREAM_TIMEOUT_S = 180.0           # float
name = "EugeneDataAgent"                  # str
is_in_conversation = req.conversation_id is not None # bool
session_id = None                         # None
include_tools = [ToolRequestEnum.EUGENE]  # list
event = {"type": "tool_call", "tool": "fetch_drug"} # dict
seen_tool_ids: set[str] = set()           # set

```

dict is the lingua franca

Almost every message between agent, tool, MCP server, and UI is a `dict`. Get fluent with these operations:

```

d = {"type": "tool_call", "tool": "fetch_drug"}

d["tool"]                # 'fetch_drug'          - raises KeyError if missing
d.get("tool_id")         # None                  - never raises
d.get("tool_id", "n/a")  # 'n/a'                - default if missing
"tool" in d              # True
d["new_key"] = 7         # add or overwrite
list(d.keys())           # ['type', 'tool', 'new_key']
list(d.values())         # ['tool_call', 'fetch_drug', 7]
for k, v in d.items():   # iterate both
    print(k, v)

```

The streaming chunk shape in Eugene's agent is exactly a `dict` — re-read this from [eugene_data_agent.py](#):

```

yield {"type": "content", "content": event["data"], "session_id": conversation_id}

```

That is *just a dict literal*. The whole streaming protocol is "yield dicts; the UI parses dicts."

Chapter 0.3. Variables — names, not boxes

```

x = 10
y = x      # y also points at 10
x = 20     # x now points at 20; y still points at 10

```

A variable in Python is a **name** bound to a value. Assignment does not copy. Mutate-vs-rebind is the most common bug source for newcomers:

```

a = [1, 2, 3]
b = a                # SAME list, two names
b.append(4)
print(a)              # [1, 2, 3, 4] ← surprise!

c = a
c = c + [5]           # NEW list assigned to c
print(a)              # [1, 2, 3, 4] unchanged

```

This matters in agents: if you share a list across conversations, one conversation mutating it pollutes the others. That is exactly the bug the AGT-01 fix in `eugene_data_agent.py` prevents:

```

def _new_conversation_manager(self) -> SlidingWindowConversationManager:
    # AGT-01 fix: instantiate per-conversation, not shared across concurrent sessions
    return SlidingWindowConversationManager(window_size=10, ...)

```

A fresh object per call → no aliasing → no leak.

Chapter 0.4. Functions — the unit of work

```

def greet(name):
    return "hello, " + name

message = greet("Rajesh")    # "hello, Rajesh"

```

Anatomy:

Piece	What it is
<code>def</code>	keyword that starts a function definition
<code>greet</code>	the function name — how callers refer to it
<code>(name)</code>	the parameter list — values the caller supplies
<code>:</code>	starts the body; the body must be indented
<code>return</code>	the keyword that sends a value back to the caller
<code>greet(...)</code>	a function call — runs the body and substitutes the return value

If you omit `return`, the function returns `None` implicitly.

What does "calling a function" actually mean?

A *call* is "stop running here, jump into that function with these inputs, run it, take its return value, and continue where I was." In Eugene this happens at four very different scales — but the concept is the same:

Scale	Eugene example	Caller	Callee
Function call	<code>_add_default_tools(chat_request.include_tools)</code>	your code	another Python function
HTTP call	<code>requests.get(_PUBMED_BASE + "/esearch.fcgi")</code>	your process	NCBI server, over the network
Tool call	agent decides <code>"call fetch_drug(id='X')"</code>	the LLM	a <code>@tool</code> function or MCP server
LLM call	<code>agent(user_prompt)</code>	your code	OpenAI/Anthropic/Bedrock model

All four are "request → response with a value." That is the unifying mental model of agentic systems.

Positional vs keyword arguments

```
def search_pubmed(query, max_results=5):
    ...

search_pubmed("emicizumab")           # positional
search_pubmed("emicizumab", 10)       # positional
search_pubmed(query="emicizumab", max_results=10)  # keyword — best for readability
```

Eugene uses **keyword arguments almost everywhere** — it is self-documenting:

```
response = eugene_agent.execute(
    token=token,
    user_prompt=prompt,
    conversation_id=chat_request.conversation_id,
    include_tools=tools,
)
```

A reader instantly knows what each argument is.

Default arguments — the mutable-default trap

```
def bad(items=[]):          # NEVER do this
    items.append(1)
    return items

bad() # [1]
bad() # [1, 1] ← the same list is reused!
```

Use `None` and create a fresh list inside:

```
def good(items=None):
    items = list(items or [])
    items.append(1)
    return items
```

Or, as Eugene does, use an *immutable* default:

```
default_tools = frozenset({...}) # frozenset is immutable → safe
```

Chapter 0.5. The `_underscore` naming conventions

Name	Meaning
name	Normal public name
<code>_name</code>	"Internal to this module/class — don't touch from outside"
<code>__name</code>	"Strongly private; Python will name-mangle it inside classes"
<code>__name__</code>	"Dunder" — a method or attribute Python itself uses
<code>name_</code>	A trailing underscore avoids clashing with a keyword (<code>class_</code> , <code>id_</code>)

You see all of these in Eugene:

```
def _add_default_tools(request_tools): ... # leading _ → "private helper"
def _manage_conversation_id(req): ...    # private to the router
agent.tool_names                         # public attribute
agent.__class__                         # dunder — set by Python itself
if __name__ == "__main__":              # dunder — true when file is the entry poi
    main()
```

The leading underscore is a convention, not enforcement. Python will let you call `_add_default_tools` from anywhere — but the underscore says "you are reaching into someone's drawer; do not be surprised if it breaks."

Chapter 0.6. Dunder methods — how Python sees your objects

A *dunder* (double-underscore) method customises how Python's built-in operations work on your objects.

```
class Extraction:
    def __init__(self, entities=set(), relationships=set()):
        self.entities = entities
        self.relationships = relationships

    def __eq__(self, other):
        # makes `a == b` work
        return self is other or (
            isinstance(other, self.__class__)
            and self.entities == other.entities
            and self.relationships == other.relationships
        )

    def __hash__(self):
        # lets the object live in a set or dict key
        return hash((self.entities, self.relationships))

    def __repr__(self):
        # what `print(x)` and the debugger show
        return "<Extraction {}:{}>".format(self.entities, self.relationships)
```

This is the real [Extraction](#) class from Eugene. Without `__eq__`, comparing two `Extraction` objects with `==` would just check identity (`is`), not contents. Without `__hash__`, you could not put one in a `set()`. Without `__repr__`, `print(extraction)` would show useless `<Extraction object at 0x10e...>`.

Other duunders you will meet:

Dunder	Triggers
<code>__init__</code>	<code>MyClass(...)</code> constructor
<code>__call__</code>	<code>obj(...)</code> — makes object callable
<code>__iter__</code>	<code>for x in obj:</code>
<code>__enter__</code> / <code>__exit__</code>	<code>with obj:</code> context manager
<code>__len__</code>	<code>len(obj)</code>

<code>__getitem__</code>	<code>obj[key]</code>	
--------------------------	-----------------------	--

Strands' Agent has `__call__` defined — that is why `agent(user_prompt)` *just works*, as if agent were a function.

Chapter 0.7. Classes — bundling state with behaviour

```
class EugeneDataAgent:
    def __init__(self, eugene_mcp_server_url: str, model):
        self.eugene_mcp_server_url = eugene_mcp_server_url
        self.model = model

    def execute(self, token, user_prompt, conversation_id, include_tools=[]):
        ...
```

Three vocabulary words:

- **Class:** the blueprint. `EugeneDataAgent` is a class.
- **Instance:** a thing you built from the blueprint. `eugene_agent = EugeneDataAgent(url, model)`.
- **Method:** a function defined inside a class. The first parameter is always `self` — the instance the method was called on.

`self.x` reads/writes attributes on **this particular instance**, not the class. Two agents can have different `self.model` values without interfering.

When to write a class vs a function

- A class makes sense when you have **state** (`model`, `mcp_server_url`) used by **several related operations** (`execute`, `execute_stream`, `_init_agent`).
- A bare function makes sense when you have **no state** — like `_add_default_tools`.

If you find yourself passing the same five arguments into every function, that is the universe whispering "make a class."

Chapter 0.8. What is Pydantic and why does Eugene love it?

Plain Python is permissive: you can pass a dict where a list is expected, and the bug surfaces five function calls later. **Pydantic** is a library that turns class definitions into *runtime-checked, JSON-serializable data containers*.

Without Pydantic

```
def chat_query(request):
    prompt = request["prompt"]          # KeyError if missing
    if not isinstance(prompt, str):      # type check by hand
        raise TypeError("prompt must be a string")
    if not prompt.strip():
        raise ValueError("prompt must not be empty")
    conversation_id = request.get("conversation_id")
    # ... 20 more lines of validation
```

Tedious, error-prone, and the *shape* of the request is buried in code.

With Pydantic (the Eugene approach)

```
# agents/eugene-agent-ws/src/query/model/chat_query_request.py
from typing import Annotated
from pydantic import BaseModel, Field
from query.model.tool_request_enum import ToolRequestEnum

class ChatQueryRequest(BaseModel):
    prompt: Annotated[str, Field(None, examples=["What assets does eugene know about biog
    conversation_id: Annotated[str, Field(None, examples=["0872c36f-7efa-438e-ac3a-d76a96
    include_tools: Annotated[list[ToolRequestEnum], Field(None, examples=[["eugene", "htt
```

Now the *class itself* declares the shape, the types, the examples. Pydantic does the validating for free. FastAPI reads the model and:

1. Parses the incoming JSON.
2. Validates every field against its type.
3. Returns a precise 422 error if anything is wrong — pointing at the exact bad field.
4. Produces an OpenAPI schema at `/docs` so the frontend team knows *exactly* what to send.

Three Pydantic concepts to commit to memory

1. A model is a class that inherits from `BaseModel`.

```
class ChatQueryRequest(BaseModel):
    prompt: str
```

2. Types are enforced.

```
ChatQueryRequest(prompt=123)
# ValidationError: prompt → Input should be a valid string
```

3. Models are JSON-aware in both directions.

```

req = ChatQueryRequest(prompt="hello")
req.model_dump()           # {'prompt': 'hello', ...}           - to dict
req.model_dump_json()      # '{"prompt": "hello", ...}'         - to JSON string
ChatQueryRequest.model_validate({"prompt": "hi"})               # dict → object
ChatQueryRequest.model_validate_json('{"prompt": "hi"}')       # JSON → object

```

That is the whole punchline: **Pydantic is the bridge between the JSON world (HTTP, LLM messages, MCP) and the Python world (typed objects you can rely on).** Every external boundary in Eugene is a Pydantic model.

Chapter 0.9. What is a "message", and what types are there?

Once you start talking to an LLM, you stop thinking in *prompts* and start thinking in *messages*. A message is a `dict` (or a Pydantic model) with three core fields:

```

{"role": "user", "content": "What are biogen's recent patents?"}

```

The **role** decides how the model treats the content.

Role	Sender	What it means to the model	Eugene example
system	the developer	"Who you are and what your rules are." Read once at the start.	The big <code>system_prompt</code> in <code>EugeneDataAgent</code>
user	the end user	"Here is what I asked."	The <code>prompt</code> field of <code>ChatQueryRequest</code>
assistant	the model itself	What the model previously said. Gives it short-term memory.	Replayed by <code>SlidingWindowConversationManager</code>

tool	a tool the model called	"Here is the result of the tool you asked for."	Output of <code>search_pubmed</code> , or any MCP tool

A full agentic conversation looks like this list of messages:

```
messages = [
    {"role": "system", "content": "You are Eugene, an agent that answers biomedical..."},
    {"role": "user", "content": "What patents has Roche filed on emicizumab?"},
    {"role": "assistant", "content": "I should look this up.",
     "tool_calls": [{"id": "t1", "name": "search_patents_web", "args": {"role": "tool", "tool_call_id": "t1", "content": "{...JSON from the tool...}"},
    {"role": "assistant", "content": "Roche filed 12 patents on emicizumab since 2018. He
]
```

This is **the whole shape of agentic Python**. Every Strands/LangChain/Bedrock call is "send this list of messages, get one or more new messages back."

Where Eugene materialises each role

- The **system message** is the `system_prompt` string in [eugene_data_agent.py](#).
- The **user message** is built from `chat_request.prompt` in [chat_query_agent_router.py](#).
- The **assistant** and **tool messages** are produced by the Strands Agent loop and stored by `FileSessionManager` under the conversation id.
- The UI receives a flattened *stream of events* — `{"type": "content" | "tool_call" | "tool_result" | "done"}` — that is **not** the message list itself, but a UI-friendly projection of it.

Memorise the distinction:

Messages are the LLM's input/output protocol. **Events** are what the UI consumes. The agent translates between them.

Chapter 0.10. Lists, sets, frozensets, sorted

```
include_tools = ["eugene", "http"] # list - ordered, duplicates allowed
inferred = set(include_tools) # set - unordered, unique
default_tools = frozenset(...) # frozenset - immutable set
sorted_tools = sorted(inferred, # list - sorted by a key function
                      key=lambda t: t.value)
```

Pick by question:

Question	Pick
"Do I need order?"	list
"Do I need uniqueness?"	set
"Do I need it to never change?"	frozenset
"Do I need both, with custom order?"	sorted(..., key=...)

Real Eugene code:

```
def _add_default_tools(request_tools: list) -> list:
    default_tools = frozenset({})          # immutable, hashable
    request = set(request_tools)           # dedupe
    return list(request.union(default_tools)) # back to a list for the API contract
```

Three different collection types in three lines, each chosen on purpose.

Chapter 0.11. Comprehensions — the one-liner that replaces a loop

```
# instead of:
names = []
for t in tools:
    names.append(t.value)

# write:
names = [t.value for t in tools]
```

Variants:

```
[t.value for t in tools if t != ToolRequestEnum.HTTP] # list with filter
{t.value for t in tools}                               # set
{t.value: 0 for t in tools}                             # dict
(t.value for t in tools)                                # generator (lazy)
```

In `chat_query_agent_router.py`:

```
f"intent: tools={ [t.value for t in tools] } reason={intent.reason}"
```

That `[t.value for t in tools]` is exactly the textbook pattern.

Chapter 0.12. Truthiness — the silent pitfall

In Python, every value has a boolean interpretation:

Falsy	Truthy
False	True
None	any non-zero int
0, 0.0	any non-empty str
" "	any non-empty list/dict/set
[], {}, set()	any object that defines <code>__bool__</code> returning True

So `if items:` is shorthand for "if there are any items." Eugene uses this everywhere:

```
if not inferred:
    inferred.add(ToolRequestEnum.EUGENE)
    reasons.append("default to graph")
```

`not inferred` is True when the set is empty. Cleaner than `len(inferred) == 0`.

But watch out: `if x:` is **not** the same as `if x is not None:`. A request with `prompt=""` is *falsy*, even though the field is present. That is why Eugene uses both styles deliberately:

```
is_in_conversation = (
    chat_query_request.conversation_id is not None
    and not chat_query_request.conversation_id == ""
)
```

Two checks, because *missing* and *empty* mean different things to the API.

Chapter 0.13. Errors and exceptions

```
try:
    response = eugene_agent.execute(...)
except Exception as e:
    logger.warning(e)
    raise e
```

- `try:` runs the protected block.
- `except SomeError as e:` catches; `e` is the error object.

- `raise` re-throws to the caller.
- `finally`: (not shown here) runs *always* — use it for cleanup.

Rule of thumb in Eugene: catch *narrow* exceptions where you know how to handle them (`requests.RequestException`, `httpx.TimeoutException`). Catch broad `Exception` *only at the edges*, where logging + re-raise is the entire response.

This is what you see in the router: catch, log, re-raise so FastAPI turns it into a 500.

Chapter 0.14. `import` — bringing names into your file

```
import json                                # whole module
from typing import Annotated, Optional    # specific names
from pydantic import BaseModel, Field     # specific names
```

Three rules:

1. Imports go at the top of the file.
2. Standard-library imports first, then third-party, then your own code.
3. `from X import *` is almost never correct — it pollutes the namespace.

You will also see *local* imports inside functions:

```
async def generate_chat_response(...):
    import json    # local import — avoids a circular import or speeds up startup
    ...
```

Use sparingly; the top of the file is the normal place.

Chapter 0.15. Putting it all together — a 30-line Eugene tour

If you understand every line below, you are ready for Part I:

```
from typing import Annotated
from pydantic import BaseModel
from fastapi import APIRouter, Depends

class ChatQueryRequest(BaseModel):          # 0.8 Pydantic
    prompt: str
    conversation_id: str | None = None

router = APIRouter()                        # 9.1 FastAPI router

def _add_default_tools(tools: list) -> list: # 0.5 underscore = private
```

return list(set(tools))	# 0.10 set→list, dedupe
@router.post("/query")	# 4.3 framework decorator
async def chat_query(	# 7 async function
req: ChatQueryRequest,	# 0.8 validated automatically
token: str = Depends(get_token),	# 9.2 dependency injection
):	
if not req.prompt.strip():	# 0.12 truthiness
raise ValueError("empty prompt")	# 0.13 exception
tools = _add_default_tools([])	# 0.4 function call
return {"prompt": req.prompt,	# 0.2 dict literal
"tools": [t for t in tools]}	# 0.11 comprehension

Twenty constructs, thirty lines, every one of them you will encounter in the rest of this book.

--	--	--	--

--	--

--	--

PART I — Orientation

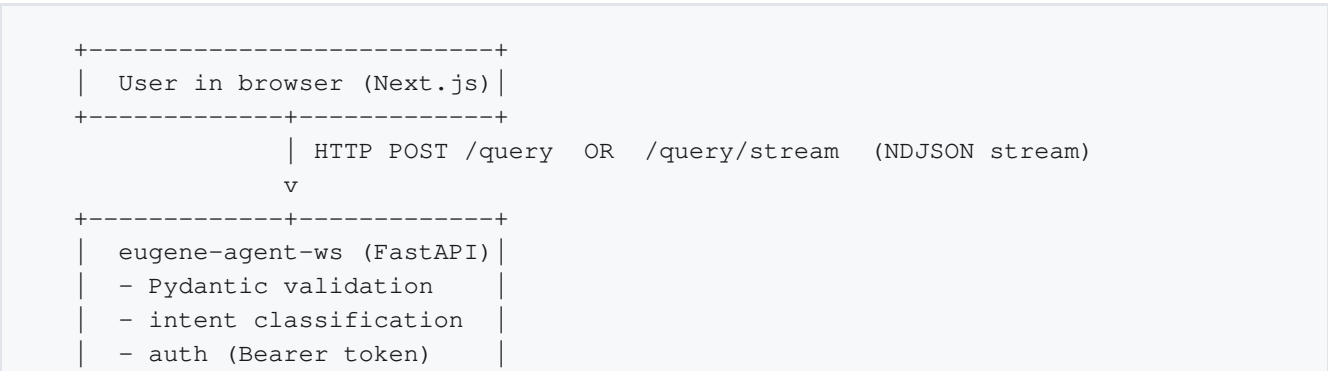
Chapter 1. Eugene at 10,000 ft

Eugene is a **biomedical competitive-intelligence platform** built around a Neo4j knowledge graph. The user talks to it in natural language; an LLM agent reasons over tools that query the graph, the web (PubMed, Google Patents), and a UI streams the reasoning trace back as it happens.

The services

Service	Path	Job
eugene-agent-ws	agents/eugene-agent-ws/	FastAPI websocket + HTTP chat agent (Strands)
eugene-mcp	agents/eugene-mcp/	MCP server exposing graph queries as tools
eugene-agent-ui / -ui-next	agents/eugene-agent-ui*	Streamlit and Next.js UIs
mlflow-0	agents/mlflow-0/	LLM evaluation harness on Bedrock + OpenAI
src/	src/	Domain libraries: organization, tpp (target product profile), graph, infra
eugene/	eugene/	Older ingestion adapters (Neo4j)
bin/docker/ ingest-*.py	bin/docker/	ETL entrypoints

The data flow (top-down)



PART II — Python foundations, grounded in Eugene

Chapter 2. Typing — the language of intent

Modern Python is *gradually typed*. Types are not enforced at runtime by default, but they are the most important documentation in the codebase. Eugene uses types everywhere.

2.1 Basic annotations

```
def _add_default_tools(request_tools: list) -> list:
    ...
```

This says: takes a list, returns a list. A reader knows the shape without reading the body.
Better:

```
def _add_default_tools(request_tools: list[ToolRequestEnum]) -> list[ToolRequestEnum]:
    ...
```

2.2 Annotated — type *plus* metadata

from typing import Annotated lets you attach metadata to a type. FastAPI and Pydantic use this for validators, dependencies, and OpenAPI hints. Eugene uses it in the route signature:

```
# agents/eugene-agent-ws/src/query/router/chat_query_agent_router.py
async def chat_query_agent(
    chat_request: Annotated[
        ChatQueryRequest, AfterValidator(validate_chat_query_request)
    ],
    token: str = Depends(get_current_token),
    user: dict = Depends(get_current_user),
):
```

Read that signature: "*chat_request is a ChatQueryRequest, **and** after parsing, run validate_chat_query_request on it.*" This is declarative validation — the function body never has to call the validator.

2.3 AsyncGenerator, Generator, Iterator

```
async def generate_chat_response(...) -> AsyncGenerator[str, None]:
```

```
...
yield json.dumps(chunk) + "\n"
```

The return type tells the reader and the type checker: *this is an async producer of `str` values; it does not accept anything via `.asend()`* (that's the `None` in the second slot).

2.4 Optional, union, the `|` shorthand

Python 3.10+ lets you write `str | None` instead of `Optional[str]`. Eugene uses both. Get comfortable with both.

2.5 `TypedDict` and dictionary shapes

A lot of the agent streaming envelope is a dictionary with a fixed shape:

```
{"type": "tool_call", "tool": "...", "tool_input": {...}, "tool_id": "..."}

```

You can describe that with `TypedDict`. Eugene currently uses plain `dict[str, Any]` for flexibility because the shape varies across LLM providers — a deliberate trade-off worth noticing.

2.6 Why typing matters here

The agent's stream comes back in shapes that differ between OpenAI and Anthropic. Look at this comment from `eugene_data_agent.py`:

```
"Strands' in-flight event shapes differ across versions/providers, but
agent.messages is stable."

```

Strong types at the boundaries (Pydantic request/response models, enum tool selection), loose types in the middle where shapes legitimately vary — that's the Eugene philosophy.

Chapter 3. Dataclasses, Enums, frozensets

3.1 Enums for closed sets

```
# agents/eugene-agent-ws/src/query/model/tool_request_enum.py
class ToolRequestEnum(str, Enum):
    EUGENE = "eugene"
    HTTP = "http"
    BIORXIV = "biorxiv"
    ...

```

Two ideas in one class:

- It is an Enum → only these values are legal; typos at the call site become exceptions.
- It inherits from str → it serializes naturally to JSON and works as a dictionary key.

Used at the boundary:

```
if ToolRequestEnum.EUGENE in include_tools:
    mcp_tools.extend(mcp_client.list_tools_sync())
```

3.2 frozenset for immutable defaults

```
default_tools = frozenset({
    # ToolRequestEnum.BIORXIV,
    # ToolRequestEnum.PUBMED
})
```

A frozenset is hashable and cannot be mutated, so it is safe as a module-level constant or a default argument. (Never use a mutable set() or [] as a default argument — a classic Python gotcha.)

3.3 Dataclasses for value objects

@dataclass autogenerates __init__, __repr__, __eq__. Eugene uses Pydantic for request/response objects (more validation), and plain dataclasses for simple internal value objects like AgentMetrics.

```
@dataclass
class AgentMetrics:
    total_tokens: int
    tool_calls: int
    duration_s: float
```

When to use which:

Need	Reach for
External JSON, validation, defaults	Pydantic BaseModel
Internal, immutable, fast	dataclass(frozen=True, slots=True)
Closed set of values	Enum
Constant set of values	frozenset

Chapter 4. Decorators — the prettiest superpower in Python

A decorator is a function that takes a function and returns a (usually wrapped) function. It is the single most important *meta-programming* tool in modern Python frameworks. Eugene uses three flavours.

4.1 Function decorator: @log_time

```
# agents/eugene-agent-ws/src/annotation/timer_annotation.py (sketch)
def log_time(func):
    @functools.wraps(func)
    async def awrap(*args, **kwargs):
        t0 = time.perf_counter()
        try:
            return await func(*args, **kwargs)
        finally:
            logger.info(f"{func.__name__} took {time.perf_counter()-t0:.3f}s")
    @functools.wraps(func)
    def swrap(*args, **kwargs):
        t0 = time.perf_counter()
        try:
            return func(*args, **kwargs)
        finally:
            logger.info(f"{func.__name__} took {time.perf_counter()-t0:.3f}s")
    return awrap if asyncio.iscoroutinefunction(func) else swrap
```

It detects sync vs async and wraps appropriately. Decorators that work on both kinds of functions are a common production pattern.

Applied:

```
@log_time
async def generate_chat_response(...):
    ...

@log_time
def _manage_conversation_id(...):
    ...
```

4.2 Library decorator: @tool (Strands)

```
# agents/eugene-agent-ws/src/query/tools/external_tools.py
from strands import tool

@tool
def search_pubmed(query: str, max_results: int = 5) -> dict[str, Any]:
```

```
"""Search PubMed for biomedical literature ..."""
```

The `@tool` decorator *introspects the function's signature and docstring* and turns it into a JSON schema the LLM sees. The docstring is not decoration — it becomes the **tool description the LLM reads to decide whether to call it**. This is a key lesson: docstrings in agent code are part of the runtime behaviour.

4.3 Framework decorators: FastAPI `@router.post`

```
@router.post("/query", response_model=ChatQueryResponse, response_model_exclude_none=True)
async def chat_query_agent(...):
    ...
```

`@router.post` registers a route. `response_model=` enforces the output shape. `response_model_exclude_none=True` strips None-valued fields from the JSON. The route function never has to think about serialization.

4.4 `@staticmethod` VS `@classmethod` VS nothing

```
class LlmFactory:
    @staticmethod
    def openai_model(openai_key, model_id="gpt-4.1-mini", ...) -> OpenAIModel:
        ...
    @staticmethod
    def anthropic_model(anthropic_key, ...) -> AnthropicModel:
        ...
```

A `LlmFactory` with only static methods is a *namespace* — used to group related functions while keeping them call-able as `LlmFactory.openai_model(...)`. There is no per-instance state.

Chapter 5. Context managers — the safe way to handle resources

`with`-statements guarantee cleanup. Eugene uses them in three crucial places.

5.1 MCP client lifecycle

```
with mcp_client:
    response = agent(user_prompt)
```

Outside this block the MCP HTTP connection is closed. The agent run is bound to the lifetime of its tool transport.

Note the subtle re-entrance trick in `_init_agent`:

```
with mcp_client:
    session_manager = FileSessionManager(session_id=conversation_id)
    ...
    mcp_tools.extend(mcp_client.list_tools_sync())
```

The same client is entered twice (once to list tools, once to run the agent). The comment in the code explicitly notes this is safe because `MCPClient` is *re-entrant*. **Lesson:** read the docs for any context manager you reuse — re-entrance is not guaranteed.

5.2 Neo4j sessions and transactions

```
# eugene/src/adapter/neo4j_adapter.py
with self.driver.session() as session:
    for node in batch:
        self._merge_node(session, node)
```

Sessions are not thread-safe; each thread or coroutine should own its own. `with` closes the session even if an exception propagates.

5.3 Building your own — `contextlib.contextmanager`

```
from contextlib import contextmanager
@contextmanager
def temp_log_level(level):
    old = logger.level
    logger.setLevel(level)
    try:
        yield
    finally:
        logger.setLevel(old)
```

Anything that "set up something, do work, tear down" should be a context manager. Promise: future-you will be grateful for the guaranteed cleanup.

Chapter 6. Generators & async generators — streaming as a language feature

A *generator* is a function that uses `yield`. Each call to `next()` runs the function until the next

`yield`, pauses, and returns the value. An *async generator* uses `yield` inside an `async def`, and is iterated with `async for`.

6.1 The chat-stream generator

```
async def generate_chat_response(token, chat_request) -> AsyncGenerator[str, None]:
    ...
    async for chunk in eugene_agent.execute_stream(...):
        yield json.dumps(chunk, ensure_ascii=False, default=str) + "\n"
```

This function does not buffer the response. Each chunk produced by the Strands agent is serialized and yielded immediately. FastAPI's `StreamingResponse` flushes it to the network. The browser sees the agent thinking *in real time*.

6.2 The timeout wrapper

```
@staticmethod
async def _with_timeout(agen: AsyncGenerator, timeout_s: float):
    loop = asyncio.get_running_loop()
    deadline = loop.time() + timeout_s
    while True:
        remaining = deadline - loop.time()
        if remaining <= 0:
            raise asyncio.TimeoutError()
        try:
            item = await asyncio.wait_for(agen.__anext__(), timeout=remaining)
        except StopAsyncIteration:
            return
        yield item
```

Read this slowly — it is one of the most instructive 12 lines in the whole codebase:

1. Compute an absolute deadline.
2. Each iteration, ask the underlying generator for one item **with a per-call timeout equal to the remaining budget**.
3. Translate `StopAsyncIteration` into a normal `return`.
4. Raise `asyncio.TimeoutError` if the deadline passes.

This is how you safely impose a *wall-clock* limit on an entire async stream that you do not own.

6.3 Generators are the easiest way to compose pipelines

```
def parse(rows):
    for r in rows: yield ...
def enrich(items):
    for it in items: yield ...
def to_neo4j(items):
    # gen — also writes
```



```
for it in items: ...
```

Then `to_neo4j(enrich(parse(read_csv(path))))` processes one record at a time, memory-bounded, regardless of file size. This is the spirit behind the ETL scripts in `bin/docker/ingest-*.py`.

PART III — Async, validation, web

Chapter 7. `asyncio` deeply: tasks, timeouts, cancellation

7.1 The mental model

`asyncio` runs a single OS thread. Coroutines voluntarily *yield* (via `await`) and the event loop picks something else to run. There is no preemption — long synchronous CPU code blocks **everything**.

7.2 Three primitives to know

Primitive	Use
<code>asyncio.gather(a, b, c)</code>	run coroutines in parallel, wait all
<code>asyncio.wait_for(coro, timeout=T)</code>	run one coroutine with a wall-clock cap
<code>asyncio.create_task(coro)</code>	fire-and-forget; you keep a handle

7.3 Eugene example: parallel external calls

If you needed to call PubMed and Google Patents in parallel inside one tool:

```
async def search_pubmed_and_patents(q):
    pubmed_task = asyncio.to_thread(search_pubmed, q, 5)  # sync→thread
    patents_task = asyncio.to_thread(search_patents_web, q)
    pubmed, patents = await asyncio.gather(pubmed_task, patents_task)
    return {"pubmed": pubmed, "patents": patents}
```

Note `asyncio.to_thread`: the `requests` library used by Eugene's web tools is synchronous. To run sync code without blocking the event loop, you push it into a thread.

7.4 Cancellation safety

When a FastAPI client disconnects mid-stream, the request task is cancelled. Inside `_with_timeout`, the `await asyncio.wait_for(...)` will raise `CancelledError`, which propagates out of the generator. *Always* leave cleanup logic in `finally`: blocks or context managers; never on the line after the `await`.

7.5 Serial vs parallel tool calls (this is the AGT-02 fix)

Inside `_init_agent` we see:

```
for attr in ("max_iterations", "max_parallel_tools"):
    try:
        setattr(agent, attr, _AGENT_MAX_ITERATIONS if attr == "max_iterations" else 1)
    except Exception:
        pass
```

`max_parallel_tools = 1` forces the agent to call tools **serially**. Why? Predictable cost, predictable ordering, simpler tool-call event extraction, and easier UI rendering. When you understand your model + tools well enough, you can bump this to call tools in parallel for latency.

Lesson: parallelism is a *capability*, but it is not a *default*. Reach for it deliberately.

Chapter 8. Pydantic v2 — validation as code

Pydantic models are the contract layer at every external boundary in Eugene.

8.1 A request model

```
class ChatQueryRequest(BaseModel):
    prompt: str
    conversation_id: str | None = None
    include_tools: list[ToolRequestEnum] = []
```

Read what this guarantees:

- `prompt` must be present and must be a `str`.
- `conversation_id` is optional.
- `include_tools` defaults to an empty list, **but every element must be a `ToolRequestEnum` value**. Anything else → 422 with a precise error path.

8.2 AfterValidator for cross-field rules

```
# agents/eugene-agent-ws/src/query/util/validation.py
def validate_chat_query_request(req: ChatQueryRequest) -> ChatQueryRequest:
    if not req.prompt.strip():
        raise ValueError("prompt must not be blank")
    return req
```

```
chat_request: Annotated[ChatQueryRequest, AfterValidator(validate_chat_query_request)]
```

`AfterValidator` runs *after* the standard parse. Use it for invariants that need the whole object (e.g. "either A or B but not both").

8.3 Field constraints

```
from pydantic import Field
class Page(BaseModel):
    size: int = Field(default=20, ge=1, le=100)
```

A 4-character expression of "between 1 and 100", validated and serialized into the OpenAPI schema FastAPI ships at `/docs`.

8.4 Why this beats hand-written checks

The validator runs *before* your route handler sees the object. Your handler can assume the object is shape-correct and focus on **what to do**, not **what to check**. That is the productivity win.

Chapter 9. FastAPI — routing, dependencies, streaming

9.1 Router pattern

```
router = APIRouter(prefix="", tags=["query"])

@router.post("/query", response_model=ChatQueryResponse, response_model_exclude_none=True)
async def chat_query_agent(...):
    ...
```

Routers are mountable. You build feature-scoped routers and then `app.include_router(query_router)` in the app entrypoint.

9.2 Dependency injection with `Depends`

```
token: str = Depends(get_current_token),
user: dict = Depends(get_current_user),
```

`get_current_token` is itself a function — possibly *also* depending on other dependencies. FastAPI walks the graph, resolves each, caches per-request, and injects. This is how Eugene plugs in OAuth/JWT verification (`router/auth/auth.py`) without polluting business logic.

9.3 Streaming responses (SSE / NDJSON)

```

return StreamingResponse(
    generate_chat_response(token, chat_request),
    media_type="text/event-stream",
    headers={"Cache-Control": "no-cache", "Connection": "keep-alive",
            "X-Content-Type-Options": "nosniff"},
)

```

Three details to copy when you do this:

- **Cache-Control: no-cache** — proxies will buffer streams without it.
- **X-Content-Type-Options: nosniff** — security best practice.
- **Newline-delimited JSON** in the body, not real SSE — Eugene uses NDJSON because it is trivial to parse on both sides while staying compatible with the `text/event-stream` mime.

9.4 Error model

In the router, exceptions are logged and re-raised. FastAPI converts them to a 500. For controlled failures use `HTTPException(status_code=400, detail="...")` explicitly. Reserve broad `try/except` for cross-cutting logging + audit; do not swallow.

PART IV — Agents, tools, models

Chapter 10. Strands Agent — the heart of Eugene

```
from strands import Agent

agent = Agent(
    name="EugeneDataAgent",
    tools=tools,
    model=self.model,
    system_prompt=self.system_prompt,
    callback_handler=debugger_callback_handler,
    conversation_manager=self._new_conversation_manager(),
    session_manager=session_manager,
)
```

Every constructor argument is a piece of architecture worth understanding.

Argument	Purpose	Eugene choice
tools	callable capabilities	MCP tools + native @tools
model	the LLM driver	OpenAI / Anthropic / Bedrock
system_prompt	role & guard rails	The big "Eugene" prompt — see §10.2
callback_handler	observe events	debugger_callback_handler — prints tool I/O for
conversation_manager	what history to keep	SlidingWindowConversationManager(window_
session_manager	where to persist history	FileSessionManager(session_id=...)

10.1 ReAct in one paragraph

ReAct = *Reason* (think), *Act* (call a tool), *Observe* (read the result), repeat. The LLM is in the loop, deciding which tool to call next based on what the previous tool returned. Strands

implements ReAct under the hood; you decide which tools are exposed and when to stop.

10.2 The system prompt is a programming language

Re-read the system prompt in `eugene_data_agent.py`. It does four things:

1. Scopes the agent ("biomedical competitive intelligence").
2. Names the tools and *when to use which*.
3. Adds **hard rules** — no `calculator` for non-math, max 20 tool calls, do not invent data, cite node ids.
4. Specifies the **output shape** — `[node_id=..., tool=...]` citation markers the UI parses.

This is a contract with the LLM. Treat prompt edits like API changes.

10.3 Anti-runaway hardening

Eugene caps:

```
_AGENT_MAX_ITERATIONS = 20
_AGENT_STREAM_TIMEOUT_S = 180
_MAX_DUPLICATE_TOOL_CALLS = 2
_HARD_TOOL_CALL_BUDGET = 25
```

...and enforces them in `_register_tool_call` mid-stream. A fingerprint `tool_name + sorted-JSON-input` detects duplicates. If exceeded, an explicit kill message is appended to the stream so the user sees *why* the loop ended. **Lesson:** every production agent loop needs at least these four caps.

Chapter 10.5. The intent classifier — how Eugene knows what you want

Before the agent ever runs, Eugene reads the user's prompt and asks: "*Which tools could possibly be relevant here?*" The result is fed into `_init_agent`, which limits the agent's toolset to only those.

Why bother? Three reasons stated explicitly in the source:

1. **Fewer tools = fewer ways for the ReAct loop to wander.** An agent can't call `current_time` if `current_time` isn't on its toolbelt.
2. **Fewer tools = shorter system prompt = lower latency and cost** — every tool description is sent to the LLM on every turn.

3. Fewer tools = easier observability — the trace is smaller and easier to debug.

10.5.1 Reading the real code

This is [intent_classifier.py](#) annotated line-by-line.

```
from __future__ import annotations          # lets us use `list[X]` syntax on older Python
import re                                   # regular expressions — Python's pattern mat
from dataclasses import dataclass          # auto __init__ / __repr__ / __eq__
from query.model.tool_request_enum import ToolRequestEnum
```

@dataclass for the return value:

```
@dataclass
class IntentResult:
    tools: list[ToolRequestEnum]
    reason: str
```

This is *not* Pydantic — it does not need runtime validation; it is internal. @dataclass is the lighter-weight choice. Calling `IntentResult(tools=[...], reason="...")` is free; you get `__init__`, `__repr__`, `__eq__` automatically.

Compiled regexes — the patterns the classifier uses:

```
_WEB_RE = re.compile(
    r"\b(web|internet|online|google|latest|recent|news|published|link|url)\b",
    re.IGNORECASE,
)
_PUBMED_RE = re.compile(
    r"\b(pubmed|publication|paper|article|literature|journal|citation|abstract)\b",
    re.IGNORECASE,
)
_PATENT_RE = re.compile(
    r"\b(patent|uspto|ip\b|intellectual property|assignee|patent\s*number)\b",
    re.IGNORECASE,
)
_GRAPH_RE = re.compile(
    r"\b(drug|disease|gene|protein|pathway|trial|indication|contraindication| "
    r"organization|target|neighbors?|relationship|node_id|biogen|roche|cs1| "
    r"hemophilia|factor|emicizumab|afstyla|eloctate|hemlibra)\b",
    re.IGNORECASE,
)
```

Three Python concepts in one block:

1. **re.compile** builds a regex object *once* (at import time). Calling `_GRAPH_RE.search(prompt)` is then fast — no recompilation per request.
2. **\b** is a *word boundary* so "patent" matches "patent" but **not** "pateNTING" or "patently". Cheap precision.

3. The leading underscore (`_WEB_RE`) marks it as private to this module — you should not reach in from outside.

The classifier function itself:

```
def classify_intent(
    prompt: str,
    user_requested: list[ToolRequestEnum] | None = None,
) -> IntentResult:
    user = list(user_requested or [])          # safe default — see Ch 0.4 mutable-default
    inferred: set[ToolRequestEnum] = set(user) # dedupe; start from what the user explicit
    reasons: list[str] = []

    if _GRAPH_RE.search(prompt):
        inferred.add(ToolRequestEnum.EUGENE)
        reasons.append("graph-concept match")
    if _WEB_RE.search(prompt) or _PATENT_RE.search(prompt) or _PUBMED_RE.search(prompt):
        inferred.add(ToolRequestEnum.HTTP)
        reasons.append("web/pubmed/patent keyword match")

    if not inferred:                          # 0.12 truthiness — empty set is falsy
        inferred.add(ToolRequestEnum.EUGENE)
        reasons.append("default to graph")

    return IntentResult(
        tools=sorted(inferred, key=lambda t: t.value),  # stable ordering
        reason=", ".join(reasons) or "none",
    )
```

Six Python idioms in 18 lines:

Line	Idiom
<code>list[ToolRequestEnum] None = None</code>	union type + safe default
<code>list(user_requested or [])</code>	"or-else" idiom for falsy fallback
<code>inferred: set[ToolRequestEnum]</code>	type-annotated local variable
<code>if not inferred:</code>	empty-collection check via truthiness
<code>sorted(..., key=lambda t: t.value)</code>	lambda for one-line key function
<code>", ".join(reasons) or "none"</code>	join to build a string + or fallback

10.5.2 Worked examples — how various user prompts get classified

Trace through what `classify_intent` returns for several prompts:

```

classify_intent("What does Eugene know about emicizumab?")
# _GRAPH_RE matches "emicizumab"
# → IntentResult(tools=[EUGENE], reason="graph-concept match")

classify_intent("Show me recent PubMed papers on hemophilia A")
# _GRAPH_RE matches "hemophilia"
# _PUBMED_RE matches "PubMed" and "papers"
# _WEB_RE matches "recent"
# → IntentResult(tools=[EUGENE, HTTP], reason="graph-concept match, web/pubmed/patent key

classify_intent("Compare Roche and Biogen patent portfolios")
# _GRAPH_RE matches "biogen" and "roche"
# _PATENT_RE matches "patent"
# → IntentResult(tools=[EUGENE, HTTP], reason="graph-concept match, web/pubmed/patent key

classify_intent("Hello")
# nothing matches
# → IntentResult(tools=[EUGENE], reason="default to graph")

classify_intent("What is 2+2?")
# nothing matches; defaults to graph
# → IntentResult(tools=[EUGENE], reason="default to graph")
# The agent then has NO calculator tool available – so it just answers from memory.

```

That last example shows the design at its best: by *removing* the calculator from the toolset for non-arithmetic prompts, we stop a class of runaway loops at the door. The intent classifier is a **defensive filter**, not a smart router.

10.5.3 Question vs ask vs compare — does Eugene care?

The classifier is **intent-by-tool**, not **intent-by-speech-act**. Eugene does not distinguish between:

Prompt shape	Example
Question	"What does Eugene know about emicizumab?"
Ask	"Tell me about emicizumab"
Compare	"Compare emicizumab and Hemlibra"
Listing	"List Roche's hemophilia drugs"
Counting	"How many patents does Roche hold?"

...because all of them point at the **same toolset** (EUGENE). The LLM, reading the system prompt and the user message, is responsible for *how* to answer (a paragraph vs a comparison table vs a count). The classifier's job is only "which tools are even on the table."

If you ever do need speech-act classification (e.g. to route to different system prompts), the pattern is exactly the same:

```
@dataclass
class SpeechAct:
    kind: str          # "compare" | "list" | "count" | "question" | "ask"
    confidence: float

_COMPARE_RE = re.compile(r"\b(compare|vs|versus|difference between)\b", re.IGNORECASE)
_LIST_RE    = re.compile(r"\b(list|show me|enumerate|all the)\b",          re.IGNORECASE)
_COUNT_RE   = re.compile(r"\b(how many|count of|number of)\b",           re.IGNORECASE)

def classify_speech_act(prompt: str) -> SpeechAct:
    if _COMPARE_RE.search(prompt): return SpeechAct("compare", 0.9)
    if _LIST_RE.search(prompt):    return SpeechAct("list",      0.9)
    if _COUNT_RE.search(prompt):  return SpeechAct("count",     0.9)
    if "?" in prompt:             return SpeechAct("question", 0.7)
    return SpeechAct("ask", 0.5)
```

Now you have a hook point: `if act.kind == "compare": system_prompt = COMPARE_PROMPT`. This is a great extension exercise (see Exercises §M).

10.5.4 Why regex, not an LLM?

You could classify intent by *calling another LLM*. Eugene deliberately does not. Reasons documented in the source:

"This is intentionally simple. A future iteration can swap in a small classifier model, but for Eugene's well-defined domain keyword rules give >90% precision with zero extra latency or cost."

The Pythonic lesson: **start with the simplest tool that solves the problem; add ML only when measurement shows you need it**. Regex compiles in microseconds; an LLM call costs ~600 ms and a few cents. Don't pay for what you don't need.

Chapter 10.6. Entity extraction — pulling structured facts out of free text

Once a tool *returns text* (PubMed abstracts, patent titles, scraped pages), Eugene often needs to pull structured **entities** and **relationships** out so they can be ingested into the graph.

10.6.1 The data model: entities and relationships

```
# src/graph/model/extraction.py (real Eugene code)
class Extraction:
    def __init__(self, entities=set(), relationships=set()):
        self.entities = entities
        self.relationships = relationships

    def __eq__(self, other):
        return self is other or (
            isinstance(other, self.__class__)
            and self.entities == other.entities
            and self.relationships == other.relationships
        )

    def __hash__(self):
        return hash((self.entities, self.relationships))

    def __repr__(self):
        return "<Extraction {}:{}>".format(self.entities, self.relationships)
```

This is the *output schema* of every extractor. Three dunder methods (see Ch 0.6):

- `__eq__` so two extractions with the same content compare equal.
- `__hash__` so an extraction can live in a set (e.g. dedupe across documents).
- `__repr__` so logs and the debugger show useful text.

`Entity` and `Relationship` are also classes — entities have a `kind` (drug, gene, disease, organization, ...) and a `name`; relationships connect two entities with a verb like `BINDS_TO`.

10.6.2 Three extraction strategies, ranked by sophistication

Strategy A: Pattern-match against a known vocabulary.

```
KNOWN_DRUGS = {"emicizumab", "afstyla", "eloctate", "hemlibra", "factor viii"}

def extract_drug_mentions(text: str) -> set[str]:
    lc = text.lower()
    return {d for d in KNOWN_DRUGS if d in lc}
```

Fast and precise *if* you have a controlled vocabulary. Eugene uses this for the well-defined drug list visible in `_GRAPH_RE` itself.

Strategy B: Regex with capture groups.

```
import re
_PMDID_RE = re.compile(r"\bPMID:\s*(\d{6,9})\b")

def extract_pmids(text: str) -> list[str]:
    return _PMDID_RE.findall(text)
# extract_pmids("see PMID: 38123456 and PMID:38123457")
# → ['38123456', '38123457']
```

The `findall` returns the *capture group* `((\d{6,9}))`, not the whole match. This is how Eugene picks PMIDs and patent numbers out of free text.

Strategy C: LLM-based extraction (structured output).

When the text is unconstrained, ask an LLM to return JSON conforming to a Pydantic schema:

```
from pydantic import BaseModel

class ExtractedEntity(BaseModel):
    kind: str          # 'drug' | 'gene' | 'disease' | 'organization'
    name: str
    span: tuple[int, int] # offsets in the source text

class ExtractionResult(BaseModel):
    entities: list[ExtractedEntity]

# Pseudocode — actual call depends on the LLM backend
result_json = llm.structured_call(
    system="Extract biomedical entities. Return strict JSON.",
    user=text,
    schema=ExtractionResult.model_json_schema(),
)
result = ExtractionResult.model_validate_json(result_json)
```

Pydantic plays a double role here:

1. It **describes** to the LLM what JSON shape to produce (via `model_json_schema()`).
2. It **validates** the model's output before your downstream code touches it.

This is exactly how `src/graph/community/analyze/extraction_provider.py` and `src/centree/mapper/extraction_util.py` operate — schema-driven extraction.

10.6.3 Walking the user's prompt as a tiny entity extractor

Here is a complete worked example. Given the user prompt:

"Show me Roche's recent patents on emicizumab."

The intent classifier already runs first and returns `{EUGENE, HTTP}`. Inside the agent, a downstream **prompt-level entity extractor** could identify:

```
{
  "drugs":          ["emicizumab"],
  "organizations": ["Roche"],
  "speech_act":     "list",
  "time_filter":    "recent",
}
```

A toy implementation:

```

@dataclass
class PromptEntities:
    drugs: list[str]
    organizations: list[str]
    speech_act: str
    time_filter: str | None

KNOWN_ORGS = {"roche": "Roche", "biogen": "Biogen", "csl": "CSL Behring"}
KNOWN_DRUGS = {"emicizumab", "hemlibra", "afstyla", "eloctate"}

def extract_prompt_entities(prompt: str) -> PromptEntities:
    lc = prompt.lower()
    drugs = [d for d in KNOWN_DRUGS if d in lc]
    orgs = [pretty for key, pretty in KNOWN_ORGS.items() if key in lc]
    if "compare" in lc or " vs " in lc or "versus" in lc: act = "compare"
    elif "list" in lc or "show me" in lc: act = "list"
    elif "how many" in lc: act = "count"
    elif "?" in prompt: act = "question"
    else: act = "ask"
    time_filter = "recent" if "recent" in lc or "latest" in lc else None
    return PromptEntities(drugs=drugs, organizations=orgs,
                          speech_act=act, time_filter=time_filter)

```

What did we use?

- **@dataclass** for the shape of the output (Ch 3.3).
- **dict literals** as controlled vocabularies (Ch 0.2).
- **List comprehensions** for the matches (Ch 0.11).
- **in operator** for substring check (Ch 0.12).
- **Cascade of if/elif/else** for the speech act (basic control flow).

Now the agent can route differently — for instance, by appending an extracted-context message to the conversation:

```

extracted = extract_prompt_entities(prompt)
hint = (f"User mentioned drug={extracted.drugs} org={extracted.organizations} "
        f"act={extracted.speech_act} time_filter={extracted.time_filter}")
messages.append({"role": "system", "content": hint})

```

That is a **second system message**, injected just for this turn. The LLM sees concrete, structured hints and is far more likely to call the right tool with the right arguments.

10.6.4 What entity extraction is *not*

It is **not** Named Entity Recognition (NER) in the deep-learning sense unless you wire in a model. Eugene's first pass is regex + vocabulary + Pydantic — and that already covers the well-defined biomedical domain. Add a model only when measured recall demands it.

Chapter 10.7. What is a "call", precisely? — four layers, one mental model

You see the word *call* everywhere: function call, method call, API call, tool call, LLM call. They are all the same shape — request, work, response — but they happen at different layers. Get this straight once and the whole stack clicks into place.

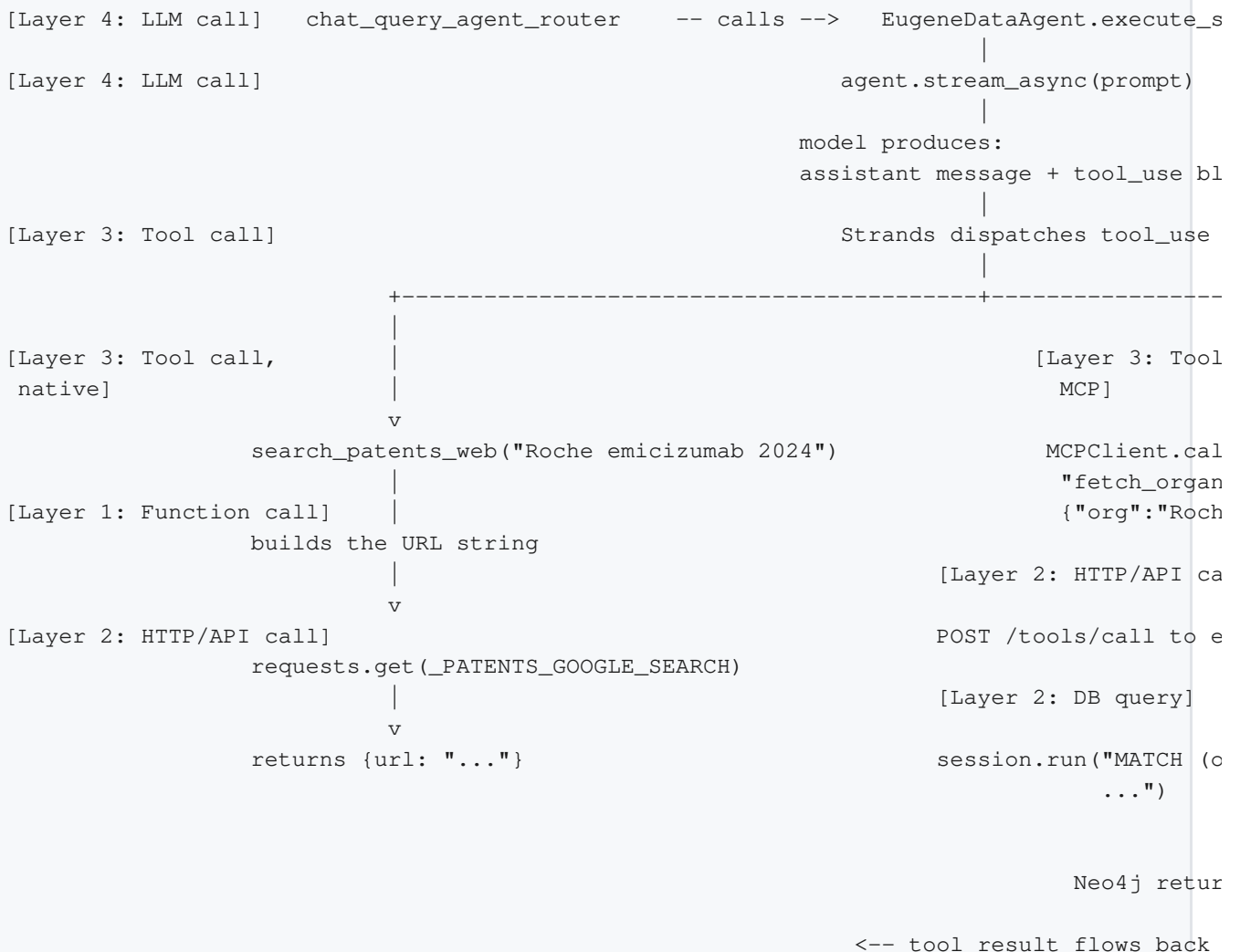
10.7.1 The four layers

Layer 1: Function call	<code>foo(x, y)</code>	- in-process, microseconds
Layer 2: HTTP/API call	<code>requests.get(...)</code>	- network, ~50-500 ms
Layer 3: Tool call	agent decides	- LLM-driven, 200 ms + tool latency
Layer 4: LLM call	agent(prompt)	- model inference, ~500-3000 ms

Each layer is *implemented in terms of* the layers below it.

10.7.2 Walk through one Eugene request

A user asks: "What patents has Roche filed on emicizumab in 2024?"



[Layer 4: LLM call ends]

```

|
model reads tool result,
produces final assistant message
|
events stream to UI
```

Read this top to bottom several times. **Everything in Eugene is one of these four call types**, nested.

10.7.3 What "the LLM calls a tool" really means

It is *not* the LLM reaching out and executing your Python. Step by step:

1. The model receives a list of messages plus a list of *tool schemas* (JSON descriptions of each `@tool`'s name, parameters, and docstring).
2. The model emits an assistant message that contains a special `tool_use` block — basically a JSON object saying "I would like to call `fetch_drug` with `{"id": "DB001"}`".
3. **Strands** sees the `tool_use` block, looks up `fetch_drug` in its tool registry, and *actually invokes the Python function* with those arguments.
4. The function returns a value. Strands wraps it in a `tool_result` message and appends it to the messages.
5. The model receives the new message list (now with the tool result) and continues — either calling another tool or producing the final assistant text.

The LLM does not "execute code." It *plans*; your runtime *executes*; the runtime feeds the result back; the LLM *observes*. That loop, repeated, is ReAct.

10.7.4 Sync call vs async call

In Python:

```
# sync call - blocks until the function returns
result = search_pubmed("emicizumab")

# async call - produces a coroutine; you must await it (or run it on an event loop)
result = await some_async_function("emicizumab")
```

Eugene exposes both, on purpose:

```
def execute(self, ...) -> AgentResult:          # sync - used by POST /query
    ...
async def execute_stream(self, ...) -> AsyncGenerator[dict, None]: # async - POST /query
    ...
```

The sync version is simpler and fine for "fetch and return one answer." The async version is

required when you want to stream events to the UI as they happen, so the user sees the agent thinking. **Pick sync for batch jobs, async for any user-facing latency-sensitive path.**

10.7.5 Serial vs parallel calls

- **Serial:** call A, wait, call B, wait, call C. Total time = $t_A + t_B + t_C$.
- **Parallel:** call A, B, C all at once. Total time = $\max(t_A, t_B, t_C)$.

Strands' agent setting `max_parallel_tools = 1` forces *serial* tool calls. This is Eugene's deliberate default for predictability. When you understand the failure modes you can flip it to $N > 1$ for latency wins.

In your own code, parallel async calls use `asyncio.gather`:

```
async def gather_facts(drug, target):
    drug_task = fetch_drug(drug)           # both coroutines
    target_task = fetch_target(target)
    return await asyncio.gather(drug_task, target_task)  # run both, wait for both
```

Two HTTP calls, one wall-clock cost. This is one of the highest-leverage performance moves in the entire stack.

Chapter 11. The tools layer — native, MCP, and how the LLM sees them

11.1 Native tools — @tool

```
@tool
def search_pubmed(query: str, max_results: int = 5) -> dict[str, Any]:
    """Search PubMed for biomedical literature ..."""
```

Three things the framework reads:

- **Function name** → tool name in the LLM API call.
- **Parameter names + types** → tool input schema.
- **Docstring** → human-readable description shown to the LLM in the tool list.

Test this discipline by re-writing the docstring of `search_pubmed`. The agent's *behaviour* changes — it picks the tool more or less aggressively. This is one of the most counter-intuitive aspects of agent programming.

11.2 MCP tools — over the network

MCP tools live in `agents/eugene-mcp/src/tools/`. The agent does not import their Python; it asks the MCP server "what tools do you have?" via `list_tools_sync()` and dispatches calls over the streamable HTTP transport.

This separation is the whole point of MCP:

- Tool implementation can be in any language.
- Tools can scale independently of agents.
- Tools can be authenticated, rate-limited, audited at one place.

11.3 The transport — `streamable_http_client`

```
return MCPClient(
    lambda: streamable_http_client(
        url=self.eugene_mcp_server_url,
        http_client=httpx.AsyncClient(
            verify=verify,
            headers={"Authorization": f"Bearer {token}"},
            timeout=httpx.Timeout(connect=10.0, read=60.0, write=30.0, pool=10.0),
        ),
    ),
)
```

The `token` is forwarded *into the MCP request headers* so the MCP server can authenticate the *user*, not just the agent. This is how Eugene preserves identity across the agent → MCP → graph hop.

11.4 Choosing native vs MCP

Pick native @tool when	Pick MCP when
It is glue code (URL builders, simple lookups)	It needs the graph driver / DB pool
It does not need auth context	It needs per-user auth
It will only ever serve one agent	It will be reused across teams

Eugene's `search_pubmed` and `search_patents_web` are native — they hit free public APIs and are cheap glue. Everything that touches the Neo4j graph is MCP.

Chapter 12. The LLM factory — OpenAI, Anthropic,

Bedrock

Eugene has *three* model backends. The factory pattern hides them behind a single interface.

12.1 OpenAI

```
model = OpenAIModel(  
    client_args={"api_key": openai_key, "max_retries": 3, "timeout": 60.0},  
    model_id=model_id,  
    params={"max_completion_tokens": max_completion_tokens,  
            "temperature": temp, "top_p": top_p},  
)
```

12.2 Anthropic

```
model = AnthropicModel(  
    client_args={"api_key": anthropic_key, "timeout": 60.0, "max_retries": 3},  
    model_id=model_id,  
    max_tokens=max_tokens,  
    params={"temperature": temp},  
)
```

12.3 Bedrock — two ways

Way A — direct via Strands

```
# agents/mlflow-0/src/conf/conf.py  
from strands.models import BedrockModel  
def agent_model() -> BedrockModel:  
    return BedrockModel(  
        model_id="anthropic.claude-3-7-sonnet-20250219-v1:0",  
        region_name="us-east-1",  
        temperature=0.0,  
    )
```

Way B — via LangChain's ChatBedrock

```
# src/infra/llm/llm_chat_factory.py  
from langchain_aws import ChatBedrock  
cls._BEDROCK_INSTANCE = ChatBedrock(  
    model_id="anthropic.claude-3-haiku-20240307-v1:0",  
    region_name="us-east-1",  
    model_kwargs={"temperature": 0.0, "max_tokens": 1024},  
)
```

When to pick which:

Need	Pick
Strands agent, ReAct loop	BedrockModel
LangChain chain / RAG / structured-output	ChatBedrock
Both ways possible	Strands — fewer adapters

12.4 IAM is the auth story

Bedrock does not take API keys — it uses **AWS IAM credentials** (env vars, instance profile, SSO). Eugene's containers receive credentials from the AWS metadata service. Practically: `boto3` "just works" if the role has `bedrock:InvokeModel` permission on the right model ARN.

12.5 The singleton pattern (mind the cache)

`llm_chat_factory.py` caches a Bedrock client in a class variable `_BEDROCK_INSTANCE`. This avoids reconnecting per request — Bedrock client construction is not free. **Pitfall:** if you mutate the cached instance (`model.temperature = 0.9`), you affect all callers. Treat factory-returned objects as immutable.

Chapter 12.5. Message types in agentic systems — the protocol every LLM speaks

You met messages briefly in Ch 0.9. This chapter is the full reference, with concrete Eugene snippets for each role.

12.5.1 The four roles

role	who writes it	what the model does with it
system	the developer (Eugene's <code>system_prompt</code>)	reads as the rules of the ga
user	the end-user via the UI	reads as the new request
assistant	the model itself, last turn	reads to remember what it sa
tool	the runtime, after executing a <code>tool_use</code>	reads as the result of an ac

12.5.2 System message — programming the agent's personality

The system message is set once when the agent is constructed:

```
agent = Agent(
    name="EugeneDataAgent",
    tools=tools,
    model=self.model,
    system_prompt=self.system_prompt,    # <- the system message lives here
    ...
)
```

Eugene's `system_prompt` is the multi-paragraph string at the top of `EugeneDataAgent`. Four sections you can transplant into any agent:

1. **Identity & scope** ("You are Eugene, an agent that answers biomedical...").
2. **Tool guide** (when to prefer which tool — this is *crucial* if you have >5 tools).
3. **Hard rules** ("Do NOT call calculator unless...", "Do NOT invent data").
4. **Output contract** (the `[node_id=<id>, tool=<tool_name>]` citation format).

A small rewrite of any of these *measurably* changes behaviour. Test changes against the `mlflow-0` harness, do not eyeball.

12.5.3 User message — what the human actually said

```
prompt = chat_request.prompt.strip()
response = eugene_agent.execute(token=..., user_prompt=prompt, ...)
```

That `user_prompt` becomes the `{"role": "user", "content": prompt}` message inside the model call. The `.strip()` is not cosmetic — leading/trailing whitespace can confuse small models.

You can append **multiple** user messages in one turn if you want to provide structured context:

```
messages.append({"role": "user", "content": prompt})
messages.append({"role": "user", "content": f"For reference, today's date is {today}."})
```

Most APIs allow this, but conventionally the second one is sent as a *system* message instead — see 12.5.6.

12.5.4 Assistant message — what the model said last turn

When the agent finishes a turn, the final assistant message contains its answer. When the conversation continues (same `conversation_id`), `SlidingWindowConversationManager` replays the last N messages so the model has *short-term memory*:

```
SlidingWindowConversationManager(
    window_size=10,                # at most 10 prior messages replayed
```

```

    should_truncate_results=True,      # tool outputs get summarised, not pasted whole
    per_turn=2,
)

```

Without this, every new user message would arrive at the model with **zero history**, and the LLM would not remember what was already said. Memory is *what makes a chat feel like a chat*.

12.5.5 Tool messages — observations after an action

A tool message follows every successful tool call:

```

{
  "role": "tool",
  "tool_call_id": "t1",          # links back to the assistant tool_use block
  "content": '{"id": "DB001", "name": "emicizumab", "year": 2017}',
}

```

The `tool_call_id` lets the model thread "this result is for *that* call I made." If you implement your own tool dispatcher, *do not lose the id* — pairing breaks and the model gets confused.

12.5.6 The hidden fifth role — "developer" / runtime context

OpenAI recently added a `developer` role conceptually equivalent to a high-priority system message. Bedrock and Anthropic still use a single `system` field. Pragmatically you can:

- keep one stable `system_prompt`, and
- prepend a *per-turn* `system` message with dynamic context (current date, user role, extracted entities).

Eugene does not do this today, but you saw it sketched in Ch 10.6.3 as a way to feed extracted entities to the LLM. It is a powerful pattern.

12.5.7 Putting it all together: one full turn in messages

```

[
  # ---- 1. set once ----
  {"role": "system",
   "content": "You are Eugene, an agent that answers biomedical competitive-intelligenc

  # ---- 2. memory replay (from prior turns) ----
  {"role": "user",      "content": "Who makes emicizumab?"},
  {"role": "assistant", "content": "Roche makes emicizumab (Hemlibra). [node_id=DB14962

  # ---- 3. this turn's user message ----
  {"role": "user", "content": "What patents have they filed on it since 2020?"},

  # ---- 4. agent decides to call a tool ----
  {"role": "assistant",

```

```

    "content": "Let me look this up.",
    "tool_calls": [{"id": "t1", "name": "search_patents_web",
                     "args": {"query": "Roche emicizumab patent 2020..2024"}}}],

# ---- 5. runtime executes the tool and appends the result ----
{"role": "tool", "tool_call_id": "t1",
 "content": "{\"url\": \"https://patents.google.com/?q=Roche+emicizumab+...\"}"},

# ---- 6. model produces the final answer using the tool result ----
{"role": "assistant",
 "content": "Roche has filed 8 patents on emicizumab since 2020. Here is the search 1
]

```

That **entire** list is what gets sent to the model on the next turn. Memory, tool use, and the new question all live in one flat sequence. Once you internalise this picture, every agentic framework — Strands, LangChain, LangGraph, OpenAI's Agents SDK, Bedrock Agents — collapses into a single concept.

Chapter 13. Memory & persistency

13.1 Two distinct concerns

- **Memory** = which messages does the LLM see this turn?
- **Persistency** = where do those messages live across requests?

13.2 `SlidingWindowConversationManager` — the memory

```

SlidingWindowConversationManager(
    window_size=10, should_truncate_results=True, per_turn=2,
)

```

- `window_size=10` — only the last 10 messages go into the prompt. Older ones are dropped.
- `should_truncate_results=True` — tool outputs (which can be huge) are summarised, not pasted verbatim.
- `per_turn=2` — within a single user turn, allow up to 2 ReAct steps before pruning.

Why instantiate it *per call* and not share?

“AGT-01 fix: instantiate *per-conversation*, not shared across concurrent sessions”

A shared manager would leak messages across users. **Lesson:** anything stateful in an agent must be scoped to the conversation, not the process.

13.3 `FileSessionManager` — the persistency

```
session_manager = FileSessionManager(session_id=conversation_id)
```

Writes conversation state to disk under a per-session directory. When the same `conversation_id` arrives in the next request, the agent resumes with prior context loaded. Swap this for a DB-backed manager in production (Strands ships several; you can write your own subclass).

13.4 The conversation id pattern

```
def _manage_conversation_id(req) -> uuid.UUID:
    is_in_conversation = req.conversation_id not in (None, "")
    return uuid.uuid4() if not is_in_conversation else uuid.UUID(req.conversation_id)
```

Server-generated `uuid4` on first message, echoed by the client thereafter. This is the canonical pattern — never let the client *invent* a session id, but accept whatever you previously returned.

PART V — MCP & the tool ecosystem

Chapter 14. MCP from both sides

14.1 What MCP is

The **Model Context Protocol** standardizes how an agent and a tool server talk. The agent says "list tools", "call tool X with args Y"; the server says "here are tools", "here is the result". The wire format is JSON-RPC. Transports include `stdio`, `sse`, and `streamable_http` (the one Eugene uses).

14.2 The Eugene MCP server (server side)

```
agents/eugene-mcp/src/
├─ tools/                each file defines a logical tool group
│   ├── eugene_drug_tools.py
│   ├── eugene_graph_tools.py
│   ├── eugene_organization_tools.py
│   └── ...
├─ resources/            static or computed read-only resources
├─ util/register.py       registers tools at startup
└─ auth/verifier.py       validates the Bearer token
```

A tool file looks like (sketch — pattern from the repo):

```
@register_tool(name="fetch_drug")
def fetch_drug(drug_id: str) -> dict:
    """Fetch a drug node from the Eugene KG by id."""
    with neo4j_session() as s:
        record = s.run("MATCH (d:Drug {id:$id}) RETURN d", id=drug_id).single()
        return record["d"] if record else {}
```

14.3 The agent (client side)

```
mcp_client = MCPClient(lambda: streamable_http_client(url=..., http_client=httpx.AsyncCli
with mcp_client:
    mcp_tools = mcp_client.list_tools_sync()
```

`list_tools_sync()` introspects the server. The agent now treats them as if they were locally defined `@tool` functions. The LLM sees one unified tool list and does not know (or care) which are local vs remote.

14.4 Adding your own MCP tool to Eugene

Three steps:

1. Add `fetch_my_thing(arg: str) -> dict` in `agents/eugene-mcp/src/tools/my_tool.py`.
2. Register it (the `util/register.py` pattern picks it up).
3. Restart the MCP server. Next agent run will see the new tool automatically.

There is *no* agent-side change. That is the value of MCP.

Chapter 15. Writing better native tools

15.1 Anatomy of `search_pubmed`

The function shows the recipe for any external-API tool:

1. **Guardrail input:** `if not query.strip(): return ...empty....`
2. **Bound the cost:** `n = max(1, min(int(max_results), 20)).`
3. **Two-step API call** (`esearch` → `esummary`). Each `raise_for_status()`.
4. **Normalize the output** to a fixed shape `{count, results: [...]}`.
5. **Never raise raw:** catch network errors into a structured `{"error": "..."} so the LLM can react.`

15.2 The patent search link builder

```
@tool
def search_patents_web(query: str) -> dict:
    """Build a Google Patents search link the user can follow."""
    return {"url": f"https://patents.google.com/?q={quote_plus(query)}"}
```

It does *not* scrape — it returns a link. That is correct: scraping Google Patents from a server gets you 503'd. Knowing what *not* to do is half the engineering.

PART VI — Knowledge graph & ETL

Chapter 16. Neo4j with Python — the driver, sessions, transactions

16.1 The driver is a connection pool

```
from neo4j import GraphDatabase
driver = GraphDatabase.driver(uri, auth=(user, password))
```

Construct **once** per process. The driver is thread-safe and pools connections. Closing it cleanly on shutdown is the only required hygiene.

16.2 Sessions and transactions

```
with driver.session(database="KnowledgeGraph") as session:
    record = session.run("MATCH (n:Drug {id:$id}) RETURN n", id=drug_id).single()
```

Two transaction styles:

- **Auto-commit** (`session.run(...)`) — one query, simple. Fine for reads.
- **Managed transaction** (`session.execute_write(fn)`) — retries on transient errors. Fine for writes.

The Eugene `Neo4jAdapter` uses auto-commit MERGEs in a loop. It is annotated as "deprecate: slow as it requires many transactions" — a clear hint to refactor to batched UNWIND writes in a single managed transaction.

16.3 Parameterized Cypher — always

```
# WRONG — string formatting, vulnerable to Cypher injection
session.run(f"MATCH (n:{label} {{id:'{id}'}}) RETURN n")
# RIGHT
session.run("MATCH (n:Drug {id:$id}) RETURN n", id=drug_id)
```

The only exception: `:Label` cannot be parameterized (Cypher restriction). Eugene works around this with `%` formatting — but **only after validating the label against a closed set**:

```
create_node = "MERGE (n1:`%s` { ... })" % node.label
```

If `node.label` came from user input you would have to whitelist it. In Eugene it comes from the ingestion schema, so the risk is bounded.

16.4 MERGE — the idempotent upsert

```
MERGE (n:Drug {id: $id})
  ON CREATE SET n.created_at = timestamp()
  ON MATCH SET n.last_seen = timestamp()
SET n.name = $name
```

Why ETL pipelines almost always use MERGE: they may run again. The same input should converge to the same graph state. Eugene's ingest loop reflects this — it is safe to re-run.

16.5 Building relationships

```
MATCH (d:Drug {id:$drug_id}), (t:Target {id:$target_id})
MERGE (d)-[r:BINDS]->(t)
  ON CREATE SET r.first_seen = timestamp()
SET r.affinity = $affinity
```

This is the workhorse pattern. Two MATCHes to find the endpoints, one MERGE for the edge, SET for properties.

16.6 Schema design lessons from Eugene

The KG has roughly these node labels: Drug, Disease, Gene, Protein, Pathway, Patent, ClinicalTrial, Organization. Properties are kept *flat and string-typed where possible* — that keeps the Cypher concise and the indices cheap.

Indices to create early:

```
CREATE INDEX drug_id FOR (n:Drug) ON (n.id);
CREATE INDEX organization_n FOR (n:Organization) ON (n.node_name);
```

Without these, `MATCH (n:Drug {id:$id})` does a label scan on the full table once you have millions of rows.

Chapter 17. ETL pipelines, the Eugene way

Look at `bin/docker/ingest-bio-data.py` and `eugene/src/adapters/*.py`. The shape:

```
extract → parse → map (to internal Node/Edge dataclasses) → load (Neo4j MERGE)
```

17.1 Extract

CSV/JSON/Excel readers live in `src/*/load/`. Use generators so the file is streamed.

```
def read_drugs(path):
    with open(path) as f:
        for row in csv.DictReader(f):
            yield row
```

17.2 Map — the schema boundary

The mapper is a pure function `dict → Node`. This is where you normalize ids, lowercase names, dedupe synonyms. *Keep the mapper deterministic*. If you have to call out to a network during mapping (e.g. canonicalize against an ontology), that becomes a separate enrichment stage.

17.3 Load

Batched MERGE, ideally with UNWIND:

```
UNWIND $rows AS row
MERGE (n:Drug {id: row.id})
SET    n += row.props
```

One query per batch of 1,000 rows is roughly 100× faster than one query per row. Refactoring `Neo4jAdapter` to do this is a great first PR to ship.

17.4 Idempotency, retries, audit

Every ingest run should:

1. Be idempotent (MERGE everywhere).
2. Log a count per batch.
3. Write a run record `((:IngestRun {started_at, ended_at, rows, source}))` so you can answer "what version of the graph is this?" later.

Chapter 18. Cypher worth memorising

```
// 1. Count by label
MATCH (n) RETURN labels(n)[0] AS label, count(*) ORDER BY count(*) DESC;

// 2. Patents about a target, joined to organization
MATCH (o:Organization)-[:HOLDS]->(p:Patent)-[:ABOUT]->(t:Target {symbol:$sym})
```

```

RETURN o.name, p.id, p.title LIMIT 50;

// 3. Shortest reasoning path
MATCH path = shortestPath(
  (d:Drug {id:$drug})-[*..6]-(dis:Disease {id:$dis})
) RETURN [n IN nodes(path) | n.name];

// 4. Top organisations by patent count, last 5 years
MATCH (o:Organization)-[:HOLDS]->(p:Patent)
WHERE p.year >= date().year - 5
RETURN o.name, count(p) AS n ORDER BY n DESC LIMIT 20;

```

The Eugene MCP tools wrap exactly these shapes. Read `agents/eugene-mcp/src/tools/eugene_graph_tools.py` and try to predict the Cypher before reading it.

--	--	--

--	--	--	--

--	--	--

--	--

--	--

--	--

PART VII — Productionizing & extending

Chapter 19. Observability

19.1 Logging discipline

- `logger = logging.getLogger(__name__)` at the top of every module.
- Log structured-ish: `f"conversation: {conversation_id}"`. Easy to grep.
- Use `logger.isEnabledFor(logging.INFO)` before constructing expensive log strings.

19.2 The `@log_time` decorator

You have seen it. It is the cheapest observability you will ever ship.

19.3 Callback handlers

```
agent = Agent(..., callback_handler=debugger_callback_handler)
```

The callback receives every model token, tool call, and tool result. In dev it dumps to stdout; in prod you swap it for a metrics emitter (Prometheus counters, Honeycomb spans).

19.4 The agent-metrics object

```
metrics = AgentMetrics.of(response)
logger.info(f"{conversation_id} response metrics: {metrics}")
```

Token counts and tool-call counts per conversation. Aggregate these — they are the *cost* signal that tells you whether your agent is well-behaved.

Chapter 20. Error handling & hard caps — the anti-runaway playbook

Eugene's `_register_tool_call` is worth memorising:

```
tool_call_budget["remaining"] -= 1
if tool_call_budget["remaining"] < 0:
    return "Tool-call budget exhausted ..."

fp = f"{tool}::{json.dumps(input, sort_keys=True, default=str)}"
```

```

dup_fingerprints[fp] = dup_fingerprints.get(fp, 0) + 1
if dup_fingerprints[fp] > _MAX_DUPLICATE_TOOL_CALLS:
    return f"Agent repeated `{tool}` with identical inputs ..."

```

Four cheap defences:

1. **Iteration cap** — `max_iterations`.
2. **Wall-clock cap** — `_with_timeout`.
3. **Total tool-call cap** — `_HARD_TOOL_CALL_BUDGET`.
4. **Duplicate-call detection** — fingerprint + count.

Every production agent should have at least three of these.

Chapter 21. Extending Eugene — a worked plan

You want to add a new capability: "**summarize the latest 5 patents on a given target.**"
Walk through the layers.

21.1 Where does the capability live?

Two viable designs:

- **Native tool** in `eugene-agent-ws`. Fast to write, no auth complexity.
- **MCP tool** in `eugene-mcp`. Reusable across agents; lives next to graph data.

Pick the second. It will read patents from the graph and summarise with the LLM.

21.2 Code skeleton

```

# agents/eugene-mcp/src/tools/eugene_patent_tools.py
from util.register import register_tool

@register_tool(name="summarize_patents_for_target")
def summarize_patents_for_target(target_symbol: str, k: int = 5) -> dict:
    """Return up to k recent patents about a target with a one-line summary each."""
    with neo4j_session() as s:
        rows = s.run(
            """
            MATCH (p:Patent)-[:ABOUT]->(t:Target {symbol:$sym})
            RETURN p.id AS id, p.title AS title, p.year AS year
            ORDER BY p.year DESC LIMIT $k
            """,
            sym=target_symbol, k=k,
        ).data()
        summaries = [{"id": r["id"], "year": r["year"],
                        "title": r["title"], "summary": _summarise(r["title"])} for r in rows]
    return {"target": target_symbol, "patents": summaries}

```


21.3 Expose it via the agent

You touch *no agent code*. The agent already calls `list_tools_sync()` on the MCP server at startup, so the new tool will appear.

21.4 Update the system prompt (optional)

Add one line to Eugene's system prompt under TOOL GUIDE:

```
Latest patents for a known target: use  
summarize_patents_for_target(target_symbol, k=5).
```

Without this the LLM will probably still find the tool, but the prompt line makes the pick reliable.

21.5 Test

```
curl -X POST http://localhost:8080/query \  
-H 'Authorization: Bearer <token>' \  
-d '{"prompt": "Show me 5 recent patents on emicizumab targets"}'
```

21.6 The lesson

A new feature in Eugene is *adding a tool*. The agent, the UI, the streaming layer — all are unchanged. This is what "agentic architecture" pays off in: a stable centre and a growing periphery.

Chapter 22. Skills, plugins, and what comes next

"Skills" in the Claude/Anthropic ecosystem are *prompted recipes* the agent can opt into. The Eugene MCP server can be extended to expose skills as resources:

- A skill = (name, description, system-prompt-fragment, recommended tools).
- The agent fetches `list_resources` and is told "you may also adopt the `regulatory_filing_summarizer` skill".

This pattern keeps growth additive — a skill is just data the agent reads.

A practical next-week roadmap:

1. Add a `skills/` directory under `eugene-mcp/src/resources/` with one YAML per skill.
2. Register them as MCP resources.
3. In the agent's system prompt, append "If a user prompt matches a skill description,

adopt that skill's tools and instruction fragment."

4. Measure quality with the `mlflow-0` harness — that is exactly what `agents/mlflow-0/` is for.

PART VIII.5 — Graph & vector algorithms

(theory → Eugene practice)

A truthful chapter. Two algorithm families come up constantly in knowledge-graph + retrieval systems: **centrality** (which nodes matter most?) and **HNSW** (the dominant approximate-nearest-neighbour index). Neither is currently coded in Eugene, but the surrounding plumbing — Neo4j GDS, Milvus, FastRP embeddings — is. This part covers the theory, shows what Eugene *does* use, and walks through the practical upgrade.

Chapter 23. Centrality — which nodes matter?

23.1 What centrality means

A *centrality* score assigns each node a real number reflecting how "important" it is in the graph. Important compared to what? The choice of definition matters enormously — there are four classical centralities to know:

Centrality	Intuition	Cost on N nodes / M edges
Degree	Just count edges. A drug with many disease links is "central."	$O(M)$ — trivial
Closeness	1 / average shortest path to all others. "Reachable" nodes win.	$O(N \cdot M)$ — expensive
Betweenness	Fraction of shortest paths in the graph that pass through this node.	$O(N \cdot M)$ — expensive
PageRank	A node is important if other important nodes link to it. Recursive.	$O(M \cdot \text{iters})$ — fast

In a biomedical KG like Eugene's:

- **Degree** trivially highlights "Factor VIII" or "Hemophilia A" — they connect to many drugs, trials, papers.
- **PageRank** highlights nodes that are central in the *network of references*, e.g. a foundational gene cited by many high-degree drugs.
- **Betweenness** highlights *bottleneck* nodes — remove them and the graph fragments. Useful for "structural holes" analyses.

23.2 What Eugene actually uses today

Grepping the repo for `gds.pageRank`, `gds.betweenness`, `gds.degree`: **zero hits**. The GDS plugin is loaded (see `EUGENE_COMPLETE_VALIDATION_DOCUMENT.md`) but the only graph-algorithm calls in code are:

```
# src/foundation/infra/db/adapter/neo4j_foundational_similarity_adapter.py
CALL gds.nodeSimilarity.filtered.stream('%s', {
    degreeCutoff: 1, similarityCutoff: .45, similarityMetric: 'COSINE',
    sourceNodeFilter: [n1]
}) YIELD node1, node2, similarity ...
```

```
# src/foundation/infra/db/adapter/neo4j_project_graph_adapter.py
call gds.graph.project(...)
```

That is, Eugene projects an in-memory graph and runs **node similarity** (Jaccard-like neighbour overlap) — not centrality. Why? Eugene's primary product question is *"what is similar to X?"* (drug-similarity, target-similarity), not *"what is most important?"* So similarity is the algorithm that pays.

23.3 The Pythonic shape of a GDS call

Every GDS call from Python follows the same three steps; learn it once.

```
# Step 1 – project a graph (in Neo4j memory) once per session
session.run("""
    CALL gds.graph.project(
        $name,
        $nodeProjection,                // e.g. '*' or ['Drug','Disease']
        $relProjection,                // e.g. '*' or {INTERACTS:{orientation:'N
        $config                          // {} or {nodeProperties: ['embeddings']}
    )
""", name="eugene", nodeProjection="*", relProjection="*", config={})

# Step 2 – run the algorithm on the named projection
result = session.run("""
    CALL gds.pageRank.stream($name, {maxIterations: 20, dampingFactor: 0.85})
    YIELD nodeId, score
    RETURN gds.util.asNode(nodeId).node_name AS name, score
    ORDER BY score DESC LIMIT 25
""", name="eugene").data()

# Step 3 – drop the projection when done (frees memory)
session.run("CALL gds.graph.drop($name, false)", name="eugene")
```

Three Python idioms appear:

- **Parameterised Cypher** — `$name`, `$nodeProjection` instead of f-strings (Ch 16.3).
- `.data()` materialises the stream into a list-of-dicts you can hand to the LLM.

- **Triple-quoted strings** for multi-line Cypher — easier to read than escaped `\n`.

23.4 Worked example — add a PageRank-based "important nodes" MCP tool

This is the smallest viable extension that adds centrality to Eugene.

```
# agents/eugene-mcp/src/tools/eugene_centrality_tools.py
from util.register import register_tool
from util.context import neo4j_session

@register_tool(name="top_important_nodes")
def top_important_nodes(label: str = "Drug", k: int = 10) -> dict:
    """Return the top-k most central nodes of a given label, by PageRank.

    Args:
        label: a node label such as "Drug", "Gene", "Organization".
        k: how many to return (1..50).
    """
    k = max(1, min(int(k), 50))
    name = f"pr_{label.lower()}"
    with neo4j_session() as s:
        s.run("CALL gds.graph.drop($n, false) YIELD graphName RETURN graphName", n=name)
        s.run("""
            CALL gds.graph.project($n, $label, '*')
            """, n=name, label=label)
        rows = s.run("""
            CALL gds.pageRank.stream($n, {maxIterations:20, dampingFactor:0.85})
            YIELD nodeId, score
            RETURN gds.util.asNode(nodeId).node_id AS id,
                   gds.util.asNode(nodeId).node_name AS name,
                   score
            ORDER BY score DESC LIMIT $k
            """, n=name, k=k).data()
        s.run("CALL gds.graph.drop($n, false) YIELD graphName RETURN graphName", n=name)
    return {"label": label, "k": k, "nodes": rows}
```

What you've used from earlier chapters:

- `@register_tool` (MCP — Ch 14).
- Type hints + docstring → schema the LLM reads (Ch 11.1).
- Parameter clamping `max(1, min(int(k), 50))` (Ch 15.1 guardrail input).
- Cypher with `$` parameters (Ch 16.3).

One line into Eugene's system prompt:

To rank entities by importance, use `top_important_nodes(label, k)`.

Now the agent can answer "Which targets are the most central in the hemophilia subgraph?"

23.5 When NOT to compute centrality

- Centralities recompute the whole graph. On Eugene's full KG (millions of relationships) a PageRank takes seconds; a betweenness takes minutes. **Cache the result, do not recompute per request.**
- Centrality on a *projection that filters by label* often gives more useful answers than on the full graph (e.g. "PageRank within drugs only"). Project deliberately.
- Centrality is not magic relevance — it ranks *structural* importance. For "which drugs are clinically relevant for hemophilia?" you want a graph traversal, not PageRank.

23.6 Centrality from pure Python (no GDS)

If you ever need this without Neo4j GDS — for instance, to compute on a small subgraph the agent retrieved:

```
import networkx as nx
G = nx.DiGraph()
G.add_edges_from(edges)
pr = nx.pagerank(G, alpha=0.85, max_iter=20)
deg = dict(G.degree())
bet = nx.betweenness_centrality(G, k=200)
top = sorted(pr.items(), key=lambda kv: -kv[1])[:10]
```

edges = [(src, dst), ...]
dict node -> score
dict node -> int
approximate, fast

`networkx` is already in Eugene's `requirements.txt`. For graphs up to ~100k nodes this is more than fast enough as a fallback.

Chapter 24. HNSW — how vector search actually works, and what Eugene uses

24.1 The problem ANN solves

You have a query vector q (the embedding of a user prompt or a summary) and a million candidate vectors. You want the top-10 nearest by cosine or inner-product distance. Exact nearest-neighbour requires comparing q against all million — $O(N \cdot d)$ per query. At $N = 1M$ and $d = 768$ that is ~750 ms per query. Far too slow.

Approximate Nearest Neighbour (ANN) indexes trade a tiny bit of recall for orders-of-magnitude speed. The two families that dominate production:

Family	Idea	Strengths	Weaknesses
IVF	k-means partition the space into cells; search a few cells	Memory-light; tunable	Lower recall at low <code>nprobe</code>

HNSW	Build a multi-layer graph; greedy walk from sparse to dense	Highest recall + lowest latency	Memory-hungry; slow build
-------------	---	------------------------------------	------------------------------

24.2 What Eugene uses today — IVF_FLAT, not HNSW

From [milvus_vector_adapter.py](#):

```
index_params.add_index(
    field_name="vector",
    index_type="IVF_FLAT",
    metric_type="COSINE",
    params={"nlist": embedding_dim},
)
```

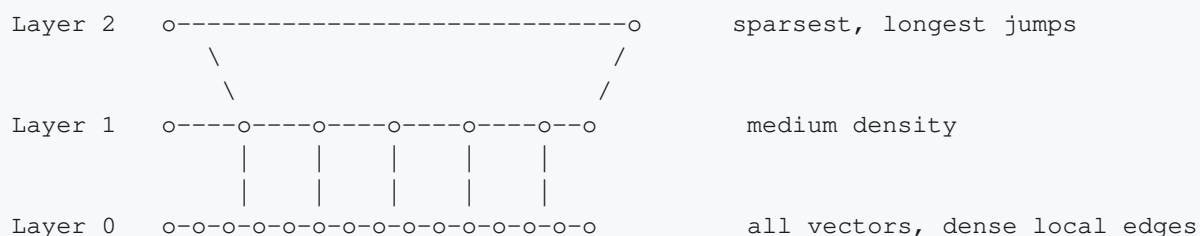
So Eugene's vector index is **IVF_FLAT** with `nlist = embedding_dim` (typically 256). Each insert is also a `metric_type="IP"` collection (inner product) — these two declarations are inconsistent in the current file and are worth tightening (see Exercise V1 below).

24.3 HNSW theory in one page

HNSW = *Hierarchical Navigable Small World*. Two ideas combined.

Idea 1 — small-world graph. Build a graph where every node (vector) is connected to its nearest $\sim M$ neighbours plus a few long-range "express" links. To find the nearest neighbour of a query q , start at any node and greedily walk to the neighbour closest to q . Repeat until no neighbour is closer. The long-range links let you "teleport" across the space in $O(\log N)$ hops instead of crawling locally.

Idea 2 — layers. Stack several such graphs:



A search descends the layers: greedy-walk on layer 2 to the rough region, drop to layer 1, refine, drop to layer 0, find the actual top-k. Each layer is $\log(N)$ times smaller than the one below, so total work is $O(\log N \cdot M)$.

Two tuning knobs you will see in every HNSW implementation:

Param	Meaning	Typical
M	edges per node in the graph	8–48
efConstruction	candidate-list size during build (higher → better graph)	100–500
ef (search)	candidate-list size during query (higher → higher recall)	50–400

Build time is roughly $O(N \cdot \text{efConstruction} \cdot M)$. Memory is $O(N \cdot M \cdot 8 \text{ bytes})$ for the graph plus the raw vectors. For $1M \times 768\text{-dim float32} + M=32$: vectors $\approx 3 \text{ GB}$, graph $\approx 256 \text{ MB}$.

24.4 IVF vs HNSW — when to switch

Pick the index based on three numbers: **N** (collection size), **QPS** (queries/sec), and **recall@10 target**.

Scenario	Pick
N < 100k, low QPS	FLAT (exact) — simplicity wins
N up to ~1M, recall@10 ≥ 0.95	IVF_FLAT (Eugene's current choice)
N > 1M, recall@10 ≥ 0.98 , latency p99 < 20 ms	HNSW
Memory-constrained, recall@10 ≥ 0.90	IVF_PQ (quantised — smaller, lossy)

Eugene's current Milvus collections are below 1M for any single label, so IVF_FLAT is a defensible default. The day you index the *entire literature corpus* across all labels, HNSW becomes worth it.

24.5 Practical: switch Milvus to HNSW in Eugene

The change is tiny because Milvus does the work. Edit `milvus_vector_adapter.py`:

```
def _create_and_list_indexes(self, collection_name: str, embedding_dim: int):
    index_params = self.client.prepare_index_params()
    index_params.add_index(field_name="id", index_type="STL_SORT")
    index_params.add_index(field_name="summary_id", index_type="INVERTED")

    # CHANGED: IVF_FLAT -> HNSW
    index_params.add_index(
        field_name="vector",
        index_type="HNSW",
        metric_type="COSINE",
```



```

        params={"M": 16, "efConstruction": 200},
    )
    self.client.create_index(
        collection_name=collection_name,
        index_params=index_params,
        sync=False,
    )

```

...and at query time, pass the search-time knob:

```

results = client.search(
    collection_name=collection_name,
    data=[query_vector],
    anns_field="vector",
    search_params={"metric_type": "COSINE", "params": {"ef": 128}},
    limit=10,
    output_fields=["summary_id", "summary"],
)

```

Two consistency fixes to do at the same time:

1. The collection is created with `metric_type="IP"` (`_ensure_collection`) but indexed with `metric_type="COSINE"` (`_create_and_list_indexes`). Pick one and align. For text embeddings normalised to unit length, IP and COSINE give the same ranking — but Milvus will complain if they disagree. Choose **COSINE** end-to-end.
2. `_ensure_collection` always drops the existing collection. That is fine for one-shot ingestion but catastrophic if called twice. Guard with an `if recreate: flag`.

24.6 HNSW from pure Python — the 30-line teaching version

You don't run this in production (you use Milvus or `hnswlib`), but reading it once burns the algorithm into your brain.

```

import math, random, heapq

class TinyHNSW:
    def __init__(self, M=8, ef=32):
        self.M, self.ef = M, ef
        self.layers = []          # layers[L][i] = list of neighbour ids
        self.vectors = []
        self.entry = None

    def _level(self):
        # pick a level with geometric prob
        return int(-math.log(random.random()) * (1/math.log(self.M)))

    @staticmethod
    def _dist(a, b):
        return sum((x-y)**2 for x, y in zip(a, b))    # squared L2

    def _search_layer(self, q, entry, layer, ef):
        visited = {entry}
        cand = [(self._dist(q, self.vectors[entry]), entry)]    # min-heap

```

```

best = list(cand) # also min-heap
while cand:
    d, n = heapq.heappop(cand)
    if d > -heapq.nsmallest(1, best)[0][0]: break # all candidates worse than
    for nb in self.layers[layer].get(n, []):
        if nb in visited: continue
        visited.add(nb)
        dn = self._dist(q, self.vectors[nb])
        heapq.heappush(cand, (dn, nb))
        heapq.heappush(best, (dn, nb))
        if len(best) > ef:
            heapq.heappop(best)
    return [n for _, n in best]

def add(self, vec):
    i = len(self.vectors); self.vectors.append(vec)
    lvl = self._level()
    while len(self.layers) <= lvl: self.layers.append({})
    if self.entry is None: self.entry = i; return
    # ... connect i to nearest M on each layer up to lvl (left as exercise) ...

```

Three Python concepts you exercise just by reading this:

- `heapq` for priority queues (`heappush`/`heappop`).
- dict of list as a sparse adjacency list — exactly how Neo4j stores edges, conceptually.
- `set()` for visited tracking — $O(1)$ membership.

24.7 The full retrieval pipeline in Eugene (with or without HNSW)

```

"show me drugs similar to emicizumab"
|
v
+-----+-----+
| sentence-transformer / model2vec embedder |
+-----+-----+
| 256-d or 768-d vector
v
+-----+-----+
| Milvus ANN search (IVF_FLAT today, HNSW |
| tomorrow). Returns top-k summary_ids. |
+-----+-----+
| summary_ids
v
+-----+-----+
| Neo4j graph enrich: MATCH (n {summary_id}) |
| RETURN node + neighbours |
+-----+-----+
| dict ready for LLM
v
agent uses it

```

This is the *retrieval-augmented* pattern. Vector store narrows the candidate set quickly; graph enriches each candidate with structure; the LLM speaks the answer. Centrality, if you add it

(Ch 23.4), becomes a re-ranker between steps 2 and 3 ("of the 50 candidates, prefer the 10 with the highest PageRank within their label").

Chapter 25. Putting it together — a smarter retrieval

A practical, end-of-book exercise that combines everything from Parts IV-VIII.

```
# A hypothetical MCP tool that fuses ANN + centrality + neighbours.
@register_tool(name="similar_with_importance")
def similar_with_importance(query_text: str, label: str = "Drug",
                            k: int = 20, top: int = 5) -> dict:
    """ANN-recall the k nearest summaries, re-rank by PageRank within `label`,
    return the top `top` with one-hop neighbours."""
    q_vec = embed(query_text) # sentence-transformer
    hits = milvus.search(q_vec, limit=k, anns_field="vector")
    cand_ids = [h["summary_id"] for h in hits]

    with neo4j_session() as s:
        # 1. centrality scores for the candidate set (compute once, cache)
        pr = {r["id"]: r["score"] for r in s.run("""
            CALL gds.pageRank.stream('eugene_main', {dampingFactor:0.85, maxIterations:20
            YIELD nodeId, score
            WITH gds.util.asNode(nodeId) AS n, score
            WHERE n.node_id IN $ids
            RETURN n.node_id AS id, score
            """, ids=cand_ids).data()}}

        # 2. fuse: cosine score * pagerank^0.5
        fused = sorted(hits,
                        key=lambda h: -h["score"] * (pr.get(h["summary_id"], 1e-6) ** 0.5))

        # 3. enrich with one-hop neighbours
        enriched = s.run("""
            MATCH (n {node_id: $id})-[r]-(m)
            RETURN n, collect({rel:type(r), node:m})[..10] AS neighbours
            """, id=fused[0]["summary_id"]).data()
    return {"query": query_text, "results": enriched}
```

What you just exercised: embedding, ANN search (whatever Milvus index is configured — IVF_FLAT or HNSW), graph centrality, fused ranking, neighbourhood enrichment, MCP tool registration. The agent gets *one* tool and inherits the full pipeline.

PART VIII — Exercises

Solving these will turn knowledge into mastery. Each one cites the file you should start in.

Easy

1. Add a new `ToolRequestEnum.ARXIV` value and wire a stub tool that returns a static result. *Start at* `query/model/tool_request_enum.py`.
2. Change the streaming response to also include an `event:` line prefix so the client can use `EventSource` instead of `NDJSON`. *Start at* `chat_query_agent_router.py`.
3. Bump `_MAX_DUPLICATE_TOOL_CALLS` to 4 via an env var. *Start at* `eugene_data_agent.py`.

Medium

1. Refactor `Neo4jAdapter.ingest` to use `UNWIND $rows ...` and benchmark against the per-row version. *Start at* `eugene/src/adapter/neo4j_adapter.py`.
2. Add a Bedrock branch to `agents/eugene-agent-ws/src/query/infra/llm/llm_factory.py` so the agent service can choose Bedrock at boot via `EUGENE_LLM_BACKEND=bedrock`.
3. Write a Pydantic model + `AfterValidator` for an "advanced search" request that requires *at least one* of `target`, `drug`, `organization`.
4. Build an MCP tool `find_recent_clinical_trials(condition, k)` and integrate it. Add a one-line entry in the system prompt.

Intent & entity (new in this edition)

M1. Extend `classify_intent` to also detect a *speech act* (compare/list/count/question).

Return a new `IntentResult.speech_act` field. *Start at* `query/util/intent_classifier.py`.

M2. Write a `PromptEntityExtractor` (Ch 10.6.3) and inject the extracted entities as an additional system message into the agent's conversation. Measure whether tool-call accuracy improves on 20 sample prompts.

M3. Replace the keyword regex in `_GRAPH_RE` with a small Bedrock-hosted Claude-Haiku call that returns a JSON list of detected biomedical entities. Compare latency, cost, and precision against the regex baseline.

Graph & vector algorithms (new in this edition)

V1. Switch Milvus from `IVF_FLAT` to `HNSW` in [milvus_vector_adapter.py](#) and benchmark `recall@10` and `p99` latency on 100k embeddings. Fix the IP-vs-COSINE inconsistency at the

same time.

V2. Implement the `top_important_nodes` MCP tool from Ch 23.4. Cache the PageRank result for 1 hour to avoid recomputation; key the cache by `(graph_name, version)`.

V3. Implement the `similar_with_importance` fused-ranking tool from Ch 25. Measure on a 20-prompt eval set whether fusing PageRank improves agreement with a human gold ranking.

V4. Add a `gds.betweennessCentrality.stream` variant alongside PageRank. Document for which user questions each is the better signal.

Hard

1. Replace `FileSessionManager` with a Postgres-backed session manager. Preserve the same external `session_id` semantics.
2. Allow the agent to call tools **in parallel** for one specific case: when summarizing N patents, fan out N `fetch_patent` calls. Keep the global serial mode otherwise. *Hint:* subclass the Strands Agent and intercept the planning step.
3. Add a `/query/replay/{conversation_id}` endpoint that re-emits the entire stream from persistence without re-invoking the LLM. *Hint:* walk `agent.messages` history.

Closing word

You started this book wanting to understand Python *through* Eugene. By now you should be able to:

- Read any file in `agents/eugene-agent-ws` and say what each construct (typing, decorator, generator, context manager, Pydantic model, FastAPI route) is doing.
- Explain the Strands Agent loop, the four anti-runaway caps, and the difference between memory and persistency.
- Compare OpenAI / Anthropic / Bedrock model backends and pick one with justification.
- Add a new MCP tool end-to-end, including the Cypher behind it.
- Design an ETL pipeline that is idempotent, batched, and observable.
- Teach this material to someone else — which is the real test of mastery.

Welcome to Eugene. Now go ship something with it.

— *End of book*