🧩 Overview

main.py is your **backend entry point** — it:

- Hosts a **FastAPI server**
- Exposes REST endpoints for each major **AI Agent**
- Orchestrates calls between **Groq**, **Python modules**, and **Rust executor**
- Handles **workflow management**, **execution**, and **error recovery**

🧱 temp file Path

```
nova-auto/
└── backend/
    └── python_core/
        ├── main.py
        ├── api/
        |   ├── planner.py
        |   ├── translator.py
        |   ├── executor.py
        |   ├── validator.py
        |   └── recovery.py
        ├── models/
        |   ├── workflow_schema.py
        |   └── command_schema.py
        └── utils/
            ├── logger.py
            ├── config.py
            └── os_helper.py
```

📘 main.py — Route Documentation

1️⃣ /plan — Command Planning Agent

Purpose:
Takes a natural-language task from the user and generates a **structured workflow JSON** using the **Groq LLM**.

Method: POST
Path: /plan

Request Body:

```
{
  "task_description": "Install Visual Studio Code",
  "target_os": "Windows"
}
```

Response Example:

```
{
  "goal": "Install VS Code",
  "commands": [
    {
      "id": "cmd_001",
      "type": "shell",
      "command": "winget install vscode",
      "os": ["Windows"],
      "retry": 2,
      "timeout": 120,
      "fallback": "gui_vscode_install"
    },
    {
      "id": "cmd_002",
      "type": "gui",
```

```
    "command": "open browser, click download, run installer",
    "os": ["Windows"],
    "retry": 1,
    "timeout": 300
   }
 ]
}
```

2️⃣ /translate — Cross-Platform Command Translator

Purpose:
Converts the generated workflow into **OS-specific implementations** (e.g., Windows → winget, macOS → brew, Linux → apt).

Method: POST
Path: /translate

Request Body:


```
{
 "workflow": { ... },
 "target_os": "macOS"
}
```


Response Example:


```
{
 "translated_workflow": {
   "goal": "Install VS Code",
   "commands": [
    {
     "id": "cmd_001",
     "type": "shell",
     "command": "brew install visual-studio-code"
    }
```

```
    ]
  }
}
```

3️⃣ /execute — Execution Layer

Purpose:
Sends a validated workflow to the **Rust executor** for command execution.

Method: POST
Path: /execute

Request Body:

```
{
  "workflow": { ... },
  "execution_mode": "auto"  // or "manual"
}
```

Response Example:

```
{
  "status": "running",
  "session_id": "exec_20251008_1242",
  "logs": []
}
```

🛂 /validate — Workflow Validation

Purpose:
Checks whether the executed workflow **succeeded** (via logs, OCR, or system state).

Method: POST
Path: /validate

Request Body:

```
{
  "session_id": "exec_20251008_1242"
}
```

Response Example:

```
{
  "cmd_001": {"status": "success"},
  "cmd_002": {"status": "skipped"},
  "cmd_003": {"status": "success"},
  "workflow_status": "completed"
}
```

5️⃣ /recover — Error Analysis & Recovery Agent

Purpose:
Analyzes failed commands and suggests **alternative steps**, **GUI fallbacks**, or **retries**.

Method: POST
Path: /recover

Request Body:

```
{
  "workflow": { ... },
  "logs": "Command failed: brew install vscode"
}
```

Response Example:

```
{
  "recovery_plan": {
    "cmd_001": {"action": "retry"},
    "cmd_002": {"action": "execute_gui_fallback"}
  }
}
```

6️⃣ /status — System & Runtime Status

Purpose:
Provides live runtime information — CPU, memory usage, active sessions, and running commands.

Method: GET
Path: /status

Response Example:


```
{
  "cpu": "14%",
  "memory": "58%",
  "active_sessions": 2,
  "running_commands": ["cmd_001", "cmd_002"]
}
```

7️⃣ /logs — Workflow Logs Viewer

Purpose:
Returns saved logs or session histories for visualization in the UI.

Method: GET
Path: /logs?session_id=exec_20251008_1242

Response Example:


```
{
  "session_id": "exec_20251008_1242",
  "logs": [
    "cmd_001: success",
    "cmd_002: fallback executed"
  ]
```

}

⚙️ **Example main.py Structure**

```
from fastapi import FastAPI
from api import planner, translator, executor, validator, recovery
from utils.logger import init_logger

app = FastAPI(title="Nova Auto Backend", version="1.0")

init_logger()

# Include modular routers
app.include_router(planner.router, prefix="/plan", tags=["Planner"])
app.include_router(translator.router, prefix="/translate", tags=["Translator"])
app.include_router(executor.router, prefix="/execute", tags=["Executor"])
app.include_router(validator.router, prefix="/validate", tags=["Validator"])
app.include_router(recovery.router, prefix="/recover", tags=["Recovery"])

@app.get("/status", tags=["System"])
async def get_status():
    return {"status": "running", "cpu": "14%", "memory": "58%"}

@app.get("/logs", tags=["System"])
async def get_logs(session_id: str):
    return {"session_id": session_id, "logs": ["cmd_001 success", "cmd_002 failed"]}

if __name__ == "__main__":
    import uvicorn
```

```
uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)
```

🧠 How It All Connects

Route	Role	Backend Module	Output
/plan	Groq LLM → JSON workflow	planner.py	Structured plan
/translate	OS adaptation	translator.py	OS-specific commands
/execute	Rust shell + GUI exec	executor.py	Execution session
/validate	Verify success	validator.py	Status summary
/recover	AI-driven fallback	recovery.py	Recovery plan
/status	System metrics	main.py	Live stats
/logs	History	main.py	Session logs