🚀 TestGen AI

"The Autonomous QA Agent from Your CLI"

A Python-based CLI package that acts as an "Autonomous QA Pair-Programmer." It lives in your terminal and automates the tedious parts of software testing: understanding code, writing test cases, running them, and formatting reports.

- **Real-Time TDD:** With **"Watch Mode,"** the tool writes tests while the user writes code, enabling true Test-Driven Development without the overhead.

2. The User Interaction (Command Logic Matrix)

This matrix defines the **Strict Separation of Concerns**. Each command has a specific job.

Feature	**testgen generate**	**testgen test**	**testgen report**	**testgen auto**
User Intent	"Create the test files for me."	"Run existing tests & show status."	"Give me a PDF/HTML document."	"Do everything (One-click)."
Logic Mode	**Creation Mode**	**Execution Mode**	**Documentation Mode**	**God Mode**
1. Analyzes Code?	✅ Yes	❌ No	❌ No	✅ Yes
2. Calls LLM?	✅ Yes	❌ No	❌ No	✅ Yes
3. Saves Files?	✅ Auto-Save	❌ No	❌ No	✅ Auto-Save
4. Runs Tests?	❌ No	✅ Yes	❌ No	✅ Yes
5. Visual Matrix?	❌ No	✅ Yes (Terminal)	✅ Yes (Terminal)	✅ Yes
6. File Report?	❌ No	❌ No (Caches data)	✅ Yes (HTML)	✅ Yes
Special Flag	--watch (Live AI)	--verbose	--pdf	N/A

3. The "AGER" Architecture

The system operates on a localized 4-step loop.

A - Analyze (The Scanner)

- Reads the directory.
- Filters noise (node_modules, .git).
- **Smart Context:** If the project is large, it extracts only function signatures/docstrings to keep LLM costs low.

G - Generate (The Brain)

- Sends context to the LLM (OpenAI/Ollama).
- Receives executable Python/Pytest code.
- Writes files to the tests/ directory.
- **Watch Mode:** Listens for file saves and triggers this step for single files instantly.

E - Execute (The Runner)

- Identifies test types (Unit vs. UI).
- Runs the actual test framework (**Pytest/Playwright**) in a subprocess.
- Captures logs and exit codes.

R - Report (The Visuals)

- Parses the execution data.
- Renders the **CLI Matrix** to the screen.
- Compiles a persistent **HTML report** for stakeholders.

4. CLI Dashboard View

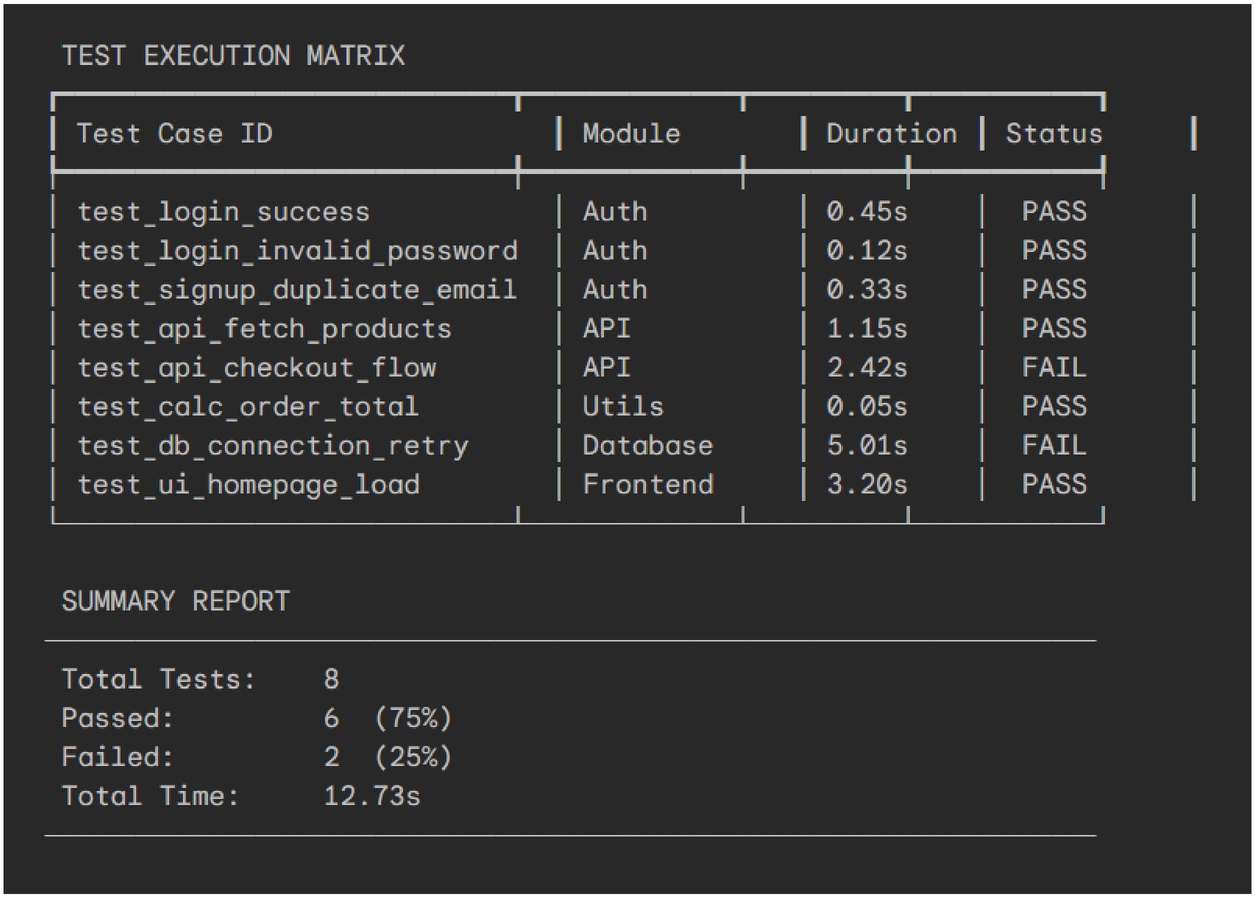
When the command finishes, the terminal output will look exactly like this:

Visual Specifications (Color Coding)

In the actual terminal, specific parts of this matrix will be colored for instant readability:

- **PASS:** Renders as **[Bold Green]** (e.g., ✔ PASS).
- **FAIL:** Renders as **[Bold Red]** (e.g., ✘ FAIL).
- **Duration:**
  - **< 1.0s:** Green (Fast).
  - **> 1.0s:** Yellow (Warning).
  - **> 5.0s:** Red (Slow/Timeout).
- **Borders:** Dimmed/Gray (to keep focus on the content).

TEST EXECUTION MATRIX



```
TEST EXECUTION MATRIX
┌──────────────────────────────┬──────────────┬──────────┬────────────┐
│ Test Case ID                  │ Module        │ Duration │ Status      │
├──────────────────────────────┼──────────────┼──────────┼────────────┤
│ test_login_success            │ Auth          │ 0.45s    │  PASS       │
│ test_login_invalid_password   │ Auth          │ 0.12s    │  PASS       │
│ test_signup_duplicate_email   │ Auth          │ 0.33s    │  PASS       │
│ test_api_fetch_products       │ API           │ 1.15s    │  PASS       │
│ test_api_checkout_flow        │ API           │ 2.42s    │  FAIL       │
│ test_calc_order_total         │ Utils         │ 0.05s    │  PASS       │
│ test_db_connection_retry      │ Database      │ 5.01s    │  FAIL       │
│ test_ui_homepage_load         │ Frontend      │ 3.20s    │  PASS       │
└──────────────────────────────┴──────────────┴──────────┴────────────┘

SUMMARY REPORT
────────────────────────────────────────────────────────────────
Total Tests:    8
Passed:         6  (75%)
Failed:         2  (25%)
Total Time:     12.73s
────────────────────────────────────────────────────────────────
```

Why this design works for your user:

- **Scannability:** The heavy use of box borders separates the data so the eyes don't get lost.
- **Triage:** The user can instantly spot the FAIL rows without reading every line.
- **Performance Auditing:** The "Duration" column helps developers spot slow tests (like that 5.01s database test above) which might need optimization.

5. Technology Stack

This stack is chosen for modularity, developer experience, and specific technical capabilities.

Component	Technology	Details	Justification
Language	Python 3.10+	Support for Pattern Matching & Type Hinting.	Standard for AI engineering & modern syntax.
CLI Framework	Typer	Uses type hints for validation & sub-commands.	Best-in-class for building modern CLIs.
Visuals	Rich	Tables, Spinners, Syntax Highlighting.	Essential for the "Matrix" visualization.
AI Layer	LiteLLM	Model Agnostic (GPT, Claude, Ollama).	Swap models without rewriting code.
Validation	Pydantic	Enforces strict JSON output from LLMs.	Prevents crashes from bad AI responses.
Observation	Watchdog	OS-level events (inotify/FSEvents).	Efficient, non-polling resource usage for --watch.
Testing Core	Pytest	Includes pytest-json-report plugin.	The execution engine for generated tests.
UI Testing	Playwright	Headless execution, auto-wait.	Superior handling of "flaky" UI tests.
Reporting	Jinja2	External template rendering.	Generates styled HTML/PDF reports.

6. File Directory Structure

The package is structured to separate **Core Logic (Brain)** from **UI Logic (Visuals)**.

```
ai-testgen/
├── pyproject.toml        # Configuration & Dependencies
├── README.md
└── src/
    └── testgen/
        ├── __init__.py
        ├── main.py        # CLI Entry Point (Typer)
        ├── manager.py     # Workflow Orchestrator (Tying it all together)
        ├── config.py      # Settings (API Keys)
        ├── core/          # THE BACKEND
        │   ├── scanner.py     # Analyzes user code
        │   ├── llm.py         # Talks to AI
        │   ├── runner.py      # Runs Pytest subprocesses
        │   └── watcher.py     # Handles --watch logic
        └── ui/            # THE FRONTEND
            ├── printer.py     # Renders the Matrix
            └── reporter.py    # Generates HTML files
```

7. Next Immediate Steps

Since the design is finalized, here is the roadmap to build the prototype:

1. **Skeleton Setup:** Create the folder structure and pyproject.toml to make the package installable.
2. **CLI Wiring:** Implement main.py with the 4 empty commands using Typer.
3. **The Scanner:** Write the logic to read a folder and return a text summary of the code.
4. **The Brain:** Connect LiteLLM to the scanner so it can actually output a "Hello World" test file.