Testing Strategy — Persistent AI Systems

Testable Units Map

Every `if/else`, `try/except`, and function parameter = a testable unit.

```
PROJECT (testable units)
│
├── temporal.py (3 units)
│   ├── Does load_and_update() calculate "5.3 hours" correctly?
│   ├── Does it survive missing/corrupt timestamps.json?
│   └── Does get_time_block() format the string correctly?
│
├── memory.py (5 units)
│   ├── Does _init_db() create the FTS5 table?
│   ├── Does add_episode() actually store data?
│   ├── Does search("college") find a memory containing "college"?
│   ├── Does search() handle special characters like quotes?
│   └── Does wipe_memory() actually clear everything?
│
├── conversation.py (4–5 units)
│   ├── Does log_message() write to the correct file?
│   ├── Does get_recent_history() return the right number of turns?
│   ├── Does buffer_clear() actually clear?
│   └── Does buffer_to_raw_text() format correctly?
│
├── renderer_base.py (4 units) ← easiest to test
│   ├── Does clean_response("[AI]: Hello") return "Hello"?
│   ├── Does validate("") return False?
│   ├── Does validate("[User]: hi") catch impersonation?
│   └── Does parse_sections() extract XML tags correctly?
│
├── packet_builder.py (integration test)
│   └── Does build() produce valid XML with all sections?
│
├── main.py (2 units)
│   ├── Does is_valid_response("") return False?
│   └── Does is_valid_response(FALLBACK_MESSAGE) return False?
│
└── PIPELINE (integration test — mock the API)
    └── input → packet_builder → renderer → validate → commit
```

How to Find Testable Units

1. **Read your code** — every `if/else`, `try/except`, and function parameter is a test case
2. **Code coverage** — `pytest --cov=. --cov-report=html` highlights untested lines in red

Example: Hidden Branches in One Function

`MemoryStore.search()` has **5 code paths** inside a single function:

Branch	Condition	Test
1	`not query or not query.strip()`	Empty input → `[]`
2	All tokens are stop words	"do you remember" → fallback to original
3	Still no tokens after fallback	→ `[]`
4	FTS5 query succeeds	Happy path
5	`sqlite3.OperationalError`	Malformed query → `[]`

Total real units across the project: **~60-80**. I listed ~20 high-value ones.

Do You Need to Find ALL?

No. Test what matters, skip the rest.

What to Test (High Value)

Category	Example
Functions with **logic** (if/else, math)	load_and_update() time delta
Functions that **touch data** (DB, files)	`memory.search()`, add_episode()
Functions that **can fail** (API, parsing)	`renderer`, parse_sections()
Public functions others depend on	`PacketBuilder.build()`
Bug-prone areas you've debugged before	Anything you've already fixed once

What to Skip (Low Value)

Category	Example
Simple getters/setters	get_time_block() return format

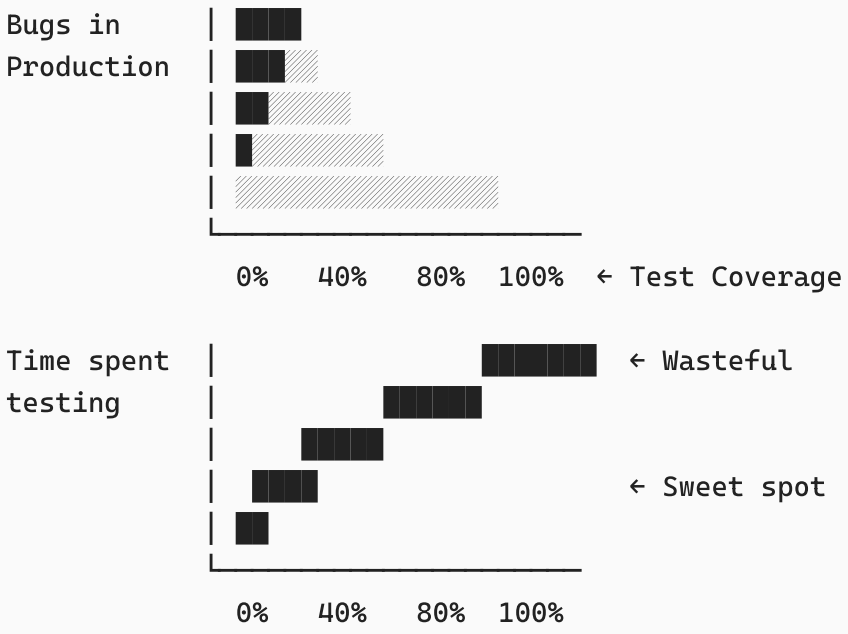
Category	Example
Print/log statements	`print("See you space cowboy...")`
Configuration constants	`CYCLE_SIZE = 5`
Code that literally can't break	`self.turn_count = 0`

Can Tests Increase?

Yes. Tests grow with your code:

- New function → new test
- New `if` condition → new branch to test
- Bug fix → **regression test** (prevents that bug from coming back)

The Testing Sweet Spot



Coverage Targets

Coverage	Who Does This	Worth It?
100%	Almost nobody	❌ Diminishing returns
90%	Google, banks, medical	For critical systems only

Coverage	Who Does This	Worth It?
80%	Industry standard	✅ **Target this**
60%	Most projects	Reasonable minimum
40%	Better than nothing	Start here
0%	Current state	❌ No confidence

The Shipping vs Testing Trade-off

> "Shipping can be delayed too much if the team wastes time on testing"

This is true **only at the extremes:**

- **0%** → **60%** = massive value, each test catches real bugs
- **60%** → **80%** = good value, catches edge cases
- **80%** → **95%** = diminishing returns, only for critical systems
- **95%** → **100%** = almost never worth it

Companies that skip testing "move fast" but spend **3x longer debugging production fire**. Companies chasing 100% on non-critical code waste engineering time.

The sweet spot: ~80% on the critical path, ~60% overall.

For This Project

Target: ~25-30 tests covering the critical path:

```
memory.search() → packet_builder.build() → renderer → validate()
```

This chain is where **95% of bugs will happen**. Test it thoroughly, and the rest falls into place.