Test Second Half

January 23, 2026

- Current project phase (Phase 1: Core, Phase 2: Advanced, Phase 3: UI, Phase 4: Production, Phase 5: Research)
- Active roadmap (72-hour research roadmap for Phase 5)
- Completed tasks (11/11 research tasks done)
- Last session summary (what was accomplished, what's next)

The state is stored in JSON files that survive crashes.

2. Git Recovery Script (`.ai_workspace/tools/recovery/recover_project.sh`)

One command recovers context:

```
bash .ai_workspace/tools/recovery/recover_project.sh
```

This script:

- Runs `git status` and `git log` to show recent changes
- Reads project state JSON to display current phase and roadmap progress
- Lists active research tasks from the roadmap tracker
- Shows the last checkpoint (if multi-agent work was interrupted)
- Outputs a recovery report: "You were working on Task LT-7, 38 of 40 hours complete, last commit was research paper v2.1 submission-ready"

This takes 2-3 seconds to run and gives complete context.

3. Roadmap Tracker (`.ai_workspace/tools/analysis/roadmap_tracker.py`)

Parses the 72-hour research roadmap markdown file and extracts:

- Total tasks (11), completed (11), in-progress (0), not started (0)
- Hours per task, total hours spent (72)
- Deliverables per task (benchmarks, proofs, papers)

This auto-updates based on git commit messages. When you commit with `feat(LT-7): Complete research paper`, the tracker marks LT-7 as done.

4. Agent Checkpoint System (`.ai_workspace/tools/checkpoints/agent_checkpoint.py`)

For multi-agent tasks that span hours, checkpoints preserve progress. Every 5-10 minutes, the agent writes a checkpoint:

- Task ID (e.g., "LT-7-research-paper")

- Agent ID (e.g., "documentation-agent")
- Progress percentage ("60% complete, methodology section done")
- Deliverables produced so far ("3 of 14 figures generated")

If interrupted, the recovery script detects incomplete checkpoints and reports: "Agent work was interrupted. Run `/resume LT-7-research-paper documentation-agent` to continue."

Together, these four components ensure no work is lost. Git preserves code. State manager preserves intent. Roadmap tracker preserves progress. Checkpoints preserve in-flight work."

Key Insights:

- The 30-second recovery time is measured. From cold start (no context), running the recovery script and reading the output takes 25-30 seconds. This is 10-100x faster than manually reading git logs, scanning files, and trying to remember what you were doing.
- Automated state tracking is essential. Manual status updates ("remember to update the roadmap file") are forgotten. Git commit message parsing is automatic – you commit code, and the roadmap updates itself.
- Multi-account recovery is a unique requirement for AI-assisted development. When using Claude Code with free tier (limited hours), you switch between multiple accounts. Without persistent state, each account starts from zero. With state in git, account B can resume account A's work seamlessly.
- Checkpoints are the safety net for long-running tasks. If an agent is generating a 40-hour research paper and crashes at hour 38, checkpoints mean you resume at hour 38, not hour 0.

Connections:

This slide connects to:

- **Section 8** – research tasks use roadmap tracker to monitor progress
- **Section 14.2** – checkpoint system details (next slide)
- **Section 20** – git workflows integrate with recovery system
- **Section 21** – future work includes expanding multi-agent orchestration

Anticipated Questions:

Q: How does git commit message parsing work?

A: "We use regex patterns to detect task IDs in commit messages. Format: `<type>(TASK-ID): <description>`. For example, `feat(LT-7): Complete research paper` matches the pattern. The roadmap tracker extracts "LT-7" and marks it as complete. This requires discipline – commit messages must follow the format – but it eliminates manual tracking."

Q: What if the JSON state files get corrupted?

A: "Git is the source of truth. If state files are corrupted or deleted, we can reconstruct them from git history. The recovery script falls back to git log analysis if JSON files are missing. It's slower (5-10 seconds instead of 2 seconds), but still works. We also backup state files to `.ai_workspace/state/backups/` weekly."

Q: Can this work for non-research projects?

A: "Absolutely. The roadmap tracker is research-specific, but the state manager and checkpoint system are generic. For a software project, you'd track sprints instead of research tasks. For

writing, you'd track chapters instead of papers. The principles – automated state tracking, git-based recovery, checkpointing long work – apply universally."

Transition:

"Session continuity handles macro-scale recovery. Now let's zoom in on the checkpoint system that handles micro-scale recovery for multi-agent tasks."

Section 17: Memory Management & Performance

[Slide 17.1] Weakref Patterns for Controller Memory Management

[Estimated speaking time: 9-11 minutes]

Context:

Memory leaks are insidious in long-running control systems. This slide explains our weakref (weak reference) pattern that prevents circular references and memory leaks in controllers, which is critical for multi-hour optimization runs and production deployment.

Main Content:

"Let's discuss memory management, which becomes critical when running PSO optimization for hours.

The problem: Python uses reference counting for garbage collection. If object A holds a reference to object B, and B holds a reference to A, you have a **circular reference**. Neither can be garbage collected because each has a non-zero reference count, even if nothing else references them. Over time, memory usage grows unbounded.

In our controllers, circular references arise naturally:

- The controller holds a history buffer (list of past states)
- Each state in the history holds metadata including a reference back to the controller
- Circular reference: controller → history → state → controller

During PSO optimization, we create thousands of controller instances (50 particles × 200 iterations × 2 evaluations = 20,000 instances). If each leaks 10 KB, that's 200 MB leaked. Unacceptable.

The solution: **weakref patterns**.

A weak reference to an object doesn't increase its reference count. If the only references to an object are weak, it can be garbage collected.

We use weakrefs in two places:

1. History Buffer

`class Controller: def` $_{i}nit_{(self):self.history=[]StrongreferencesOKhere}$
`def cleanup(self):  Explicitly clear history self.history.clear()`

Instead of holding strong references in both directions, we make the controller responsible for explicit cleanup. Before destroying a controller instance, call `controller.cleanup()` to break the circular reference.

2. Callback Registration

If controllers register callbacks with a simulation runner:

`def register`$_{c}allback(self, callback) : Storeweakreferenceinsteadofstrongself.callbacks.append(weakref.r$

This ensures that when a controller is destroyed, it doesn't stay alive because the simulation runner is holding a reference.

We validate memory management through 11 specific tests in `tests/test_integration/test_memory_manageme`
These tests:

- Create thousands of controller instances

- Verify they're garbage collected after cleanup
- Check that memory usage returns to baseline
- Test multi-threaded scenarios (concurrent creation/destruction)

Result: 11/11 memory tests passing, zero leaks detected in 10-hour PSO runs."

Key Insights:

- Python's garbage collector *will* eventually collect circular references (it has a cycle detector), but this is slow and unreliable for real-time systems. Explicit cleanup with weakrefs gives deterministic behavior.
- The weakref pattern is a trade-off: it requires discipline (remembering to call `cleanup()`) but guarantees no leaks. In production, we enforce this through quality gates – any controller without a cleanup method fails code review.
- Memory tests are essential. Without them, you don't know if you have a leak until production deployment fails after 10 hours. The tests simulate long-running scenarios (thousands of instances) to catch leaks early.
- Thread safety matters: if one thread creates a controller while another destroys one, race conditions can cause use-after-free bugs. Our weakref patterns are thread-safe through careful use of locks (tests validate this with pytest-xdist parallel execution).

Connections:

This slide connects to:

- **Section 1** – 328 Python files, all following weakref patterns
- **Section 4** – PSO creates thousands of controller instances, memory management prevents leaks
- **Section 7** – 11 memory tests validate the patterns
- **Section 12** – HIL long-running experiments benefit from leak-free controllers
- **Section 22** – performance metrics include memory usage benchmarks

Anticipated Questions:

Q: Why not just use a language with automatic garbage collection like Java?

A: "Java *does* have automatic GC, but it's non-deterministic and can cause pauses (stop-the-world GC). For real-time control, unpredictable pauses are unacceptable. Python + explicit cleanup gives us deterministic memory management. Also, the scientific Python ecosystem (NumPy, SciPy, Matplotlib) is unmatched, and rewriting 328 files in Java would take months."

Q: What tools do you use to detect memory leaks?

A: "For testing: pytest-memray, which profiles memory allocation per test. For production: tracemalloc (Python stdlib) to snapshot memory before/after PSO runs. If after-snapshot is significantly higher than before-snapshot (accounting for cached results), we have a leak. We also use heapy (guppy3 package) to inspect the heap and find what objects are accumulating."

Q: Do all 7 controllers use the same memory pattern?

A: "Yes, they share a common base class (`ControllerBase`) that implements the weakref pattern. Concrete controllers (Classical, STA, etc.) inherit this, so they automatically get correct memory management. This is why having a factory pattern (Section 1) is valuable – we enforce patterns

consistently across all implementations."

Transition:

"Memory management ensures efficiency. Now let's wrap up Part III and transition to Part IV: Professional Practice, where we discuss operational aspects like UI testing, workspace organization, and version control."