AI-Assisted Development Strategy

Structured Workflows for AI usage in the jbox Project

Jordan Vieler

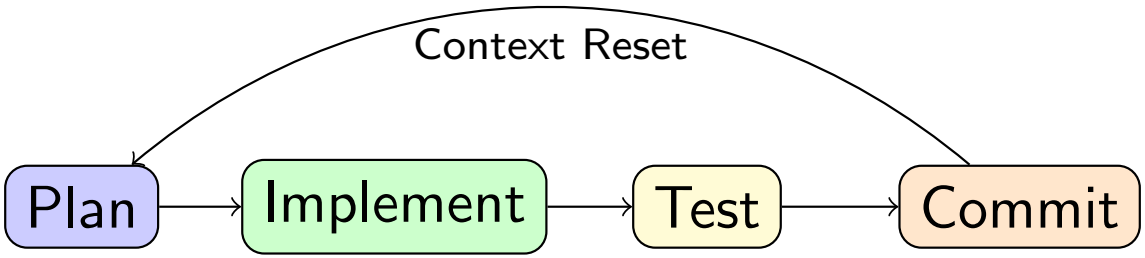
UCSC Silicon Valley Extension
EMBD.X420 - Linux Systems Programming

December 17, 2025

Outline

Development Cycle

The development process follows a **plan-implement-verify-commit** cycle that maximizes AI effectiveness while maintaining code quality.



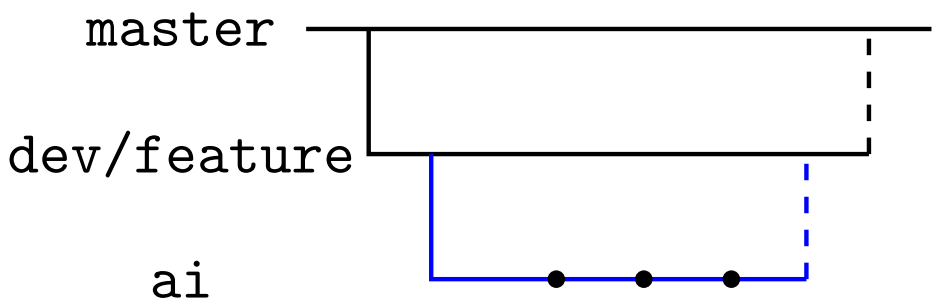
Key principle: **Incremental progress with context resets** maintains coherence across sessions.

Branching Strategy

Workflow

AI-assisted development uses a dedicated branch to isolate AI work:

1. **Create AI branch** — Branch off from current dev/feature branch
2. **AI performs work** — All implementation on the `ai` branch
3. **Merge to dev/feature** — Once complete, merge back
4. **Merge to master** — Final integration



Branching Strategy

Benefits

- **Isolation** — AI work is separated from ongoing development
- **Review opportunity** — Changes can be reviewed before merging
- **Easy rollback** — Branch can be discarded if issues arise
- **Clean history** — Squash merges consolidate AI commits

Project Setup

Establish Conventions Early

Before implementation, establish clear conventions the AI will follow:

`CONVENTIONS.md` Code style, naming, standards (C23, etc.)

`APPANATOMY.md` Standard command structure and modules

`CLItools.md` CLI specification for agent-facing commands

`CLAUDE.md` Project-level instructions for the AI

Key insight: Specification files define *target architecture* before features are implemented, ensuring coherence across the codebase.

Project Setup

Standardize Testing

Choose a single testing framework and format:

- **Python unittest** for all tests in this project
- Consistent naming: `test_<component>.py`
- Consistent structure: `tests/<category>/<component>/`
- Tests generated *alongside* implementation

Benefit

Uniform testing structure allows AI to generate tests consistently without needing repeated instructions.

Feature Implementation

Phase 1: Planning

Write a Detailed Specification:

- Feature requirements and behavior
- Input/output specifications
- Edge cases and error handling
- Integration points with existing code

Generate Implementation Plan:

- Ask AI to output `AI_TODO_<N>.md`
- Include phase breakdown with milestones
- Task checklists with `[ ]` checkboxes
- File locations and dependencies

Feature Implementation

Phase 2: Implementation

Instruct the AI to execute with clear constraints:

```
Proceed with the implementation plan in AI_TODO_<N>.md.

Constraints:
- Check off todos as they are completed
- Generate tests in the appropriate directory
- Update Makefile targets when necessary
- Follow conventions in CONVENTIONS.md
- Generate Doxygen-style docstrings
- Commit changes after each implementation phase
```

Incremental Progress: Implement → Test → Update build → Mark complete → Commit

Feature Implementation

Phase 3: Context Management

After each major phase or commit:

1. **Reset the conversation context** (start fresh session)
2. Instruct AI to continue from where it left off
3. Reference the `AI_TODO_<N>.md` file as persistent state

Why Reset Context?

- Prevents context overflow in long implementations
- Maintains fresh, focused sessions
- `AI_TODO` file serves as the "memory"

Quick Changes Workflow

For small changes that don't require extensive planning:

$$\textbf{Change} \longrightarrow \textbf{Test} \longrightarrow \textbf{Commit}$$

Characteristics:

- Single session, no plan required
- Changes made directly
- Tests run to verify
- Committed immediately

Examples: Bug fixes, documentation updates, small refactors, configuration changes

Debugging Workflow

When tests fail or bugs are detected:

1. **Reset context** if necessary (avoid confusion)
2. **Provide clear bug report:**

```
A bug has been detected in <component>.

Symptoms:
- <what is happening>

Expected behavior:
- <what should happen>

Relevant files:
- <file paths>

Test output:
<paste failing test output>
```

AI then: Analyze → Identify root cause → Fix → Verify → Commit

Documentation Standards

Doxygen-Style Docstrings

All functions include documentation:

```
/**
 * @brief Short description of the function
 *
 * Longer description if needed, explaining
 * behavior, edge cases, and important details.
 *
 * @param param1 Description of first parameter
 * @param param2 Description of second parameter
 * @return Description of return value
 */
int function_name(int param1, char *param2);
```

Documentation Standards

AI_TODO File Format

Implementation plans follow a consistent structure:

```
# Feature Implementation Plan

## Overview
Brief description of what's being implemented.

## Phase 1: Component Name
### 1.1 Task Group
- [ ] Specific task
- [ ] Another task

## Phase 2: Next Component
...

## Notes
Additional context, dependencies, considerations.
```

Best Practices

Do

- **Be specific** in requirements and constraints
- **Break large features** into multiple phases
- **Reset context** between major phases
- **Verify plans** before implementation
- **Test incrementally** as features are built
- **Commit frequently** with descriptive messages

Best Practices

Don't

- Don't let context grow too large
- Don't skip the planning phase for complex features
- Don't implement without tests
- Don't ignore failing tests
- Don't forget to update build system

Session Templates

New Feature Session

```
Implement <feature> for the jbox project.

<detailed requirements>

Output an implementation plan as AI_TODO_<N>.md
following the format in existing AI_TODO files.
```

Session Templates

Continue Implementation Session

```
Continue with the implementation plan in
ai/plans/AI_TODO_<N>.md.

Resume from Phase <X>.

Constraints:
- Check off todos as completed
- Generate tests in tests/<category>/
- Update Makefile when needed
- Use Doxygen-style docstrings
- Follow CONVENTIONS.md
- Commit after each phase
```

Session Templates

Bug Fix & Quick Change

Bug Fix:

```
Debug the following issue in <component>:

<bug description and test output>

Fix the bug and verify with tests.
```

Quick Change:

```
<describe the change needed>

Make the change, test it, and commit.
```

Workflow Summary

Scenario	Approach
New major feature	Plan → Implement → Test → Commit → Reset
Small change	Change → Test → Commit
Bug fix	Reset → Debug → Fix → Test → Commit
Documentation	Write → Review → Commit

This strategy enables efficient AI-assisted development while maintaining **code quality**, **consistency**, and **project coherence**.

Key Takeaways

Lessons Learned

1. **AI works best on well-defined incremental changes**
   - Small, focused tasks yield better results than large ambiguous ones
2. **Project structure matters**
   - Modular design allows changes to slot in cleanly
   - Tests can be added systematically
   - Build system can be easily expanded
3. **Simple, statically-typed languages help**
   - C with well-established best practices
   - AI translates specifications to code more consistently
   - Clear contracts reduce ambiguity

4. **AI struggles with contract changes**
   - Updating callers when interfaces change is error-prone
   - Since "generating" code is cheap, define first:
     1. Architecture and contracts
     2. Dependencies
     3. Tests
   - Then generate tests, then proceed with implementation

5. **Design before generation**
   - Front-load architectural decisions
   - Use specification documents as the source of truth
   - Let AI implement against a predetermined design

Key Takeaways

The Formula

Well-Defined Structure

$+$

Clear Specifications

$+$

Incremental Changes

$=$

Effective AI-Assisted Development

Questions?