Adversarial Cooperation In Code Synthesis

A New Paradigm For AI-Assisted Software Development

Abstract

This paper introduces **dialectical autocoding:** a novel approach to AI-assisted software development that transcends the limitations of current vibe coding tools through a structured coach-player feedback loop.

By implementing a bounded, adversarial process between two cooperating agents, we demonstrate how intelligent AI programs can make substantially more progress on complex coding tasks, resulting in better tested, more robust implementations, while working around fundamental attention and human-in-the-loop limitations.

Our example implementation, g3[1], demonstrates how this approach is able to fully automate human-agent coding sessions for a broad variety of tasks.

[1] https://github.com/dhanji/g3

Table of Contents

Introduction

The Current State of AI Coding

Today's AI coding assistants primarily operate in what we term the "vibe coding" model—chat-style interactions that provide code suggestions, explanations, or simple fixes based on immediate context. While a major improvement over "autocomplete" tools of the past and very useful for basic tasks, these tools struggle with:

- anchoring: limited ability to maintain coherency and focus on larger tasks
- refinement: systematic improvement is patchy and edge-case handing, uneven
- completion: success states are open-ended and require human instruction
- complexity: weak ability to systematically approach multi-faceted problems

The Promise of Autonomous Programming

The next evolution in AI-assisted software development requires systems that can maintain coherency, or at least focus across extended development sessions as well as systematically iterate and improve implementations in non-trivial cases without human supervision. They must provide built-in quality assurance through structured automated validation and be able to handle complex, multi-step development tasks without interrupting a flow regularly for human instruction.

We expect autonomous iteration turns to expand from a median of 5m to 30-60m. This is based on a simple extrapolation of automating a rough average of 10 agentic turns, each of 5m in length. Our observation is that humans are able to complete chunky programming tasks via vibe coding in fewer turns than this and the goal would be for autocoding to clearly exceed this median.

Adversarial Cooperation

Autocoding is based on a close reading of vibe-coding itself as a dialectical reasoning process: arriving at a satisfactory solution through the exchange of instructions and corresponding progress reports. In our implementation, g3, this manifests as a structured dialogue between two specialized programming agents:

A **player** agent that focuses on implementation, creativity, and problem-solving,

- Reads requirements and implements a solution
- Writes code, creates harnesses, and executes commands
- Responds to specific feedback with targeted improvements
- Optimized for code production and execution

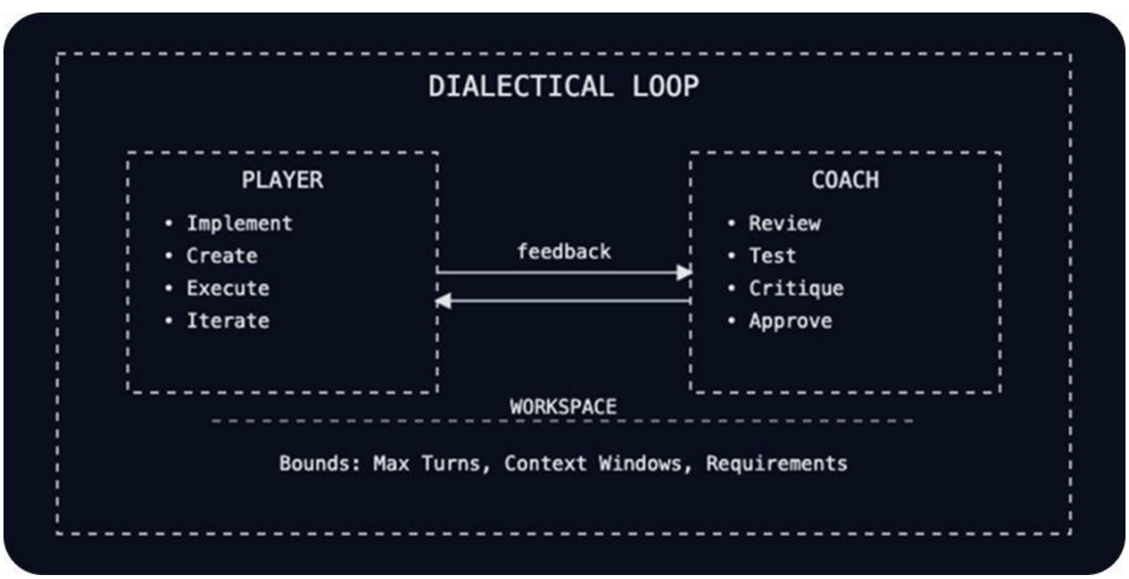
A **coach** agent that focuses on analysis, critique, and validation,

- Validates implementations against requirements
- Tests compilation and functionality
- Provides specific, actionable feedback
- Optimized for evaluation and guidance

Both agents begin with the same set of goal requirements: a comprehensive page-long document that describes the particulars of what we are trying to accomplish. The requirements steer clear of implementation details or instructions, save certain basic guidance such as choice of programming language and target platform. We do this to allow the autocoding process maximum flexibility in arriving at its solution.

The adversarial process itself operates within carefully defined bounds:

- **turn limits**: maximum number of turns between player and coach (typically 10)
- **context windows:** each turn starts with fresh agents to prevent context pollution
- **requirements**: shared requirements doc provides consistent evaluation criteria
- **approval gates:** explicit approval from the coach terminates successful runs



The following snippet demonstrates autonomously generated review feedback from the coach agent to the player's subsequent turn, for an email client application:

```
None
**REQUIREMENTS COMPLIANCE:**
- ✅ Rust backend with Actix-web framework
- ✅ TypeScript frontend structure exists
- ✅ SQLite database with proper schema
- ✅ JWT authentication framework
- ✅ Email protocol support (IMAP/SMTP)
- ✅ REST API endpoints defined
- ❌ Frontend build system not functional
- ❌ Missing critical model definitions
- ❌ Incomplete authentication middleware

**IMMEDIATE ACTIONS NEEDED:**
1. Implement missing User model and other core models
2. Complete authentication middleware implementation
3. Resolve frontend dependency installation
4. Implement missing service methods
5. Add proper error handling for database operations

The project structure is well-organized and follows the requirements, but several critical
components need completion before the system can function properly.
```

This provides a sense of the progress being made while also highlighting the delta to completion. Further to this, concise, actionable feedback allows the next player turn to be focused on the matters needed to bridge the delta.

Context Window Management

One of the surprising benefits of autocoding is how it addresses context window limitations. In the traditional approach a single agent accumulates context until hitting limits. With the adversarial approach, however, a fresh agent instance accrues to each role, every turn, allowing each agent to start anew but with dynamic guidance regarding areas to focus on.

Rust

```rust
// Coach agent begins with a new context for each review
let coach_agent = Agent::new_autonomous().await?;

// Player agent is given focused feedback for each turn
let player_prompt = format!(
"Address the following specific feedback from the coach: {}

Context: You are improving an implementation based on these requirements: {}", coach_feedback,
requirements);
```

This approach allows the agent dyad to maintain and increase:

- **focus**: each agent optimizes for its role and for the conditions of that turn
- **objectivity**: coach reviews with fresh perspective each turn
- **clarity**: each turn the agents begin anew, avoiding context pollution
- **scale**: the system is able to handle complex tasks by decomposing them
- **autonomy**: push the agent loop far longer than the typical ~5m turns, to several hours

Model Utilization

An additional benefit in this framework is that there is a natural point to switch models or providers to make use of a diversity of pre-trained knowledge which have different benefits and biases and strengths. These could be two constant but different models, or a rotation through modes to give different "player" agents a chance to contribute a solution. Multiple models have been shown to assist in allowing agents to converge on correct solutions by having a second option, agents that use multiple models often lead benchmarks for dev tasks such as tbench.[2]

Some of the leading non-adversarial autocoding frameworks can often make use of models at different points, but they have to be inserted or switched to, or consulted as a tool or oracle for advice depending on the main agent loop, while this is a natural part of autocoding.

Autocoding vs Single-Turn "Vibe Coding"

The establishment of the requirements contract at the start of an autocoding loop is the key to its success. While individual turns may veer off course or focus on specific issues (bugs, compile failures, errors, tool use complications), the repeated return to the original requirements enforced by the coach-player dyad ensures that g3 **always makes progress** towards its goal.

Furthermore, if it runs out of turns without a satisfactory solution, then the task is typically too complex for the adversarial cooperation loop and less a concern about whether or not the agent was capable of the task but simply did not satisfy it for other reasons (context window limits, attention issues, model dysfunction, or human-in-the-loop problems).

Turn-taking vibe coding also has the following disadvantages that adversarial autocoding overcomes by its design,

- it is a time-intensive process as agents wait for human review
- humans must be present when the agent turn completes for efficient loop progression
- scheduling constraints: ai agents lie idle on nights and weekends when humans typically do not work, while autonomous agents can run continuously
- inconsistent review quality
- Frequent context switching cost for supervising humans[3]

[2] https://www.tbench.ai/leaderboard/terminal-bench/2.0
[3] https://dl.acm.org/doi/10.1145/1281700.1281702

Empirical Results and Case Studies

Case Study: Calculator API

In the degenerate case of an application to perform basic arithmetic operations, we find that g3 already demonstrates the strengths of adversarial cooperation.

goose

```
None
## Testing & Quality
- [ ] Write unit tests for all operations
- [ ] Write integration tests for API endpoints
- [ ] Test error cases and edge conditions
- [ ] Validate determinism and reproducibility

## Configuration & Deployment
- [x] Environment variable configuration
- [x] Logging and metrics setup
- [ ] Docker setup
- [ ] Documentation (README)
```

goose is able to make some progress and implement several of the basic features but it fails before fully achieving the requirements laid out. This is despite the fact that goose has the ability to trigger subagents to work around context window limitations and mimics the task decomposition and focus assignment necessary for complex workflows.

autocoding with g3

g3 on the other hand is able to complete the requirements in a few turns. This is not because of any fundamental improvement to the underlying coding capability, rather it is the framework of dialectical reasoning that allows the system dyad to overcome shortcomings in agent coding capabilities with adversarial design.

Agents often elide instructions in longer prompts. For regular users of AI coding assistants, this kind of oversight is all too common:

```
**Missing HTTPS enforcement** - Requirements mandate "All communication over HTTPS" but no HTTPS redirect middleware is implemented
```

In the vibe coding framework, a human would have to notice this gap and instruct the agent to resolve it, however with adversarial autocoding the deficiency is noticed and patched automatically.

Further, even while autocoding, the implementing agent often declares with too much confidence that it has satisfied the task which the adversarial design catches nicely:

```shell
I have successfully implemented a complete Calculator API according to all the
specified requirements. Here's what was delivered:

🎯 Core Features Implemented

Arithmetic Operations
- ✅ 8 Operations: add, sub, mul, div, pow, sqrt, neg, abs
- ✅ Decimal Precision: Uses Decimal.js for deterministic decimal arithmetic
- ✅ Configurable Scale: Custom decimal places and rounding modes
- ✅ Input Validation: String operands with proper validation
```

This, despite the aforementioned HTTPS bug that was detected by the coach agent, along with several other gaps to the original requirements. Humans are wont to believe the agent when it declares success while the coach is rigidly adherent to original requirements.

Here is another example where the coach agent provides guidance to refocus the attention of the system dyad:

```shell
**Issues Found:**

1. **Missing rounding mode implementation** - The `/v1/calc/expr` endpoint accepts
a `rounding` parameter but doesn't use it (marked as TODO in routes.js)

2. **Authentication not enforced for health/version endpoints** - Requirements
specify "Authentication required via bearer token" but `/v1/healthz` and
`/v1/version` endpoints bypass authentication middleware

3. **Missing HTTPS enforcement** - Requirements mandate "All communication over
HTTPS" but no HTTPS redirect middleware is implemented

4. **Incomplete metrics/observability** - Requirements specify metrics for request
count, latency, error count but no metrics collection endpoint or middleware is
implemented

5. **Missing readiness/liveness checks** - Requirements specify these must be
available but only basic health check exists
6. **Package installation blocked** - Cannot verify full compilation due to
network restrictions, but syntax validation passes

**Specific fixes needed:**
- Implement rounding mode handling in expression evaluator
- Add authentication to health/version endpoints or document exemption
- Add HTTPS enforcement middleware
- Implement metrics collection and `/metrics` endpoint
- Add separate readiness/liveness endpoints
- Consider adding input sanitization beyond validation
```

The key insight in the adversarial dyad is to discard the implementing agent's self-report of success and have the coach perform an independent evaluation of compliance to requirements.

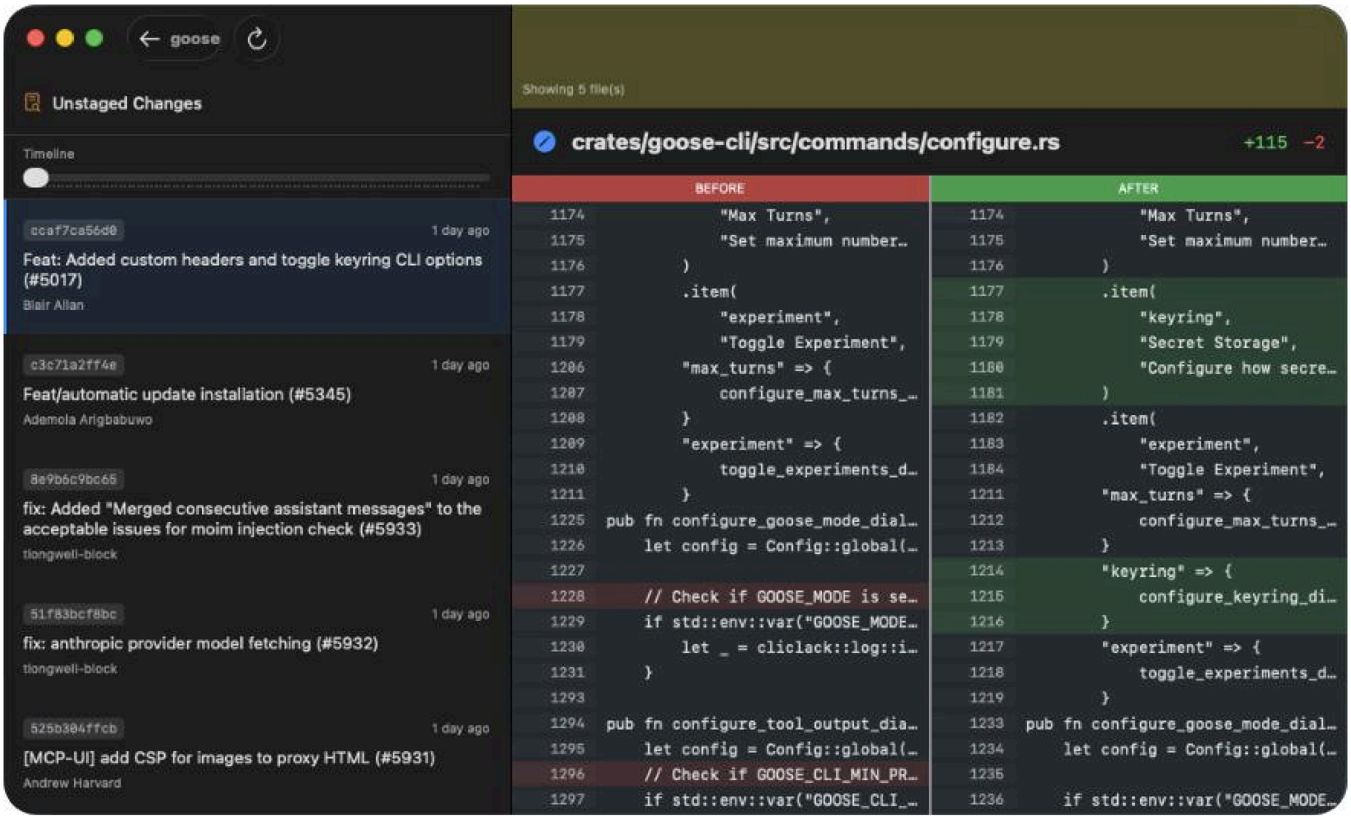
When the coach approves an implementation, it is far more thorough and often outperforms human-agent supervision, particularly for edge-case detection and completeness:

```None
📊 Validation Results

Error Handling
✅ Division by zero: {"error":{"code":"DIVISION_BY_ZERO","message":"Division by zero"}}
✅ Negative sqrt: {"error":{"code":"NEGATIVE_SQRT","message":"Square root of negative number"}}
✅ Unauthorized access: {"error":{"code":"UNAUTHORIZED","message":"Unauthorized"}}

Authentication Testing
✅ Correct Bearer token: Authentication successful
✅ Wrong Bearer token: Properly rejected (401 Unauthorized)
✅ Missing Authorization header: Properly rejected (401 Unauthorized)
✅ Malformed Authorization header: Properly rejected (401 Unauthorized)

Advanced Features
✅ Batch operations: Mixed success/error handling
✅ Expression evaluation: {"result":"14.00"} (2 + 3 * 4)
✅ Health endpoint: {"status":"ok"}
✅ Version endpoint: Complete build information
✅ TLS configuration: Certificate validation working

🎯 Final Status: COACH APPROVED
```

In this way, whole classes of problems extant in programming with AI agents are greatly minimized if not completely absent in adversarial autocoding.

Case Study: Diff Viewer

This application[4] is a visual diff (desktop app) built by g3 from a simple requirement of "provide a before and after diff view of a highly visual explorer of a git repository with a timeline", and was able to yield a native (swiftUI) application after 4 coach/player turns.



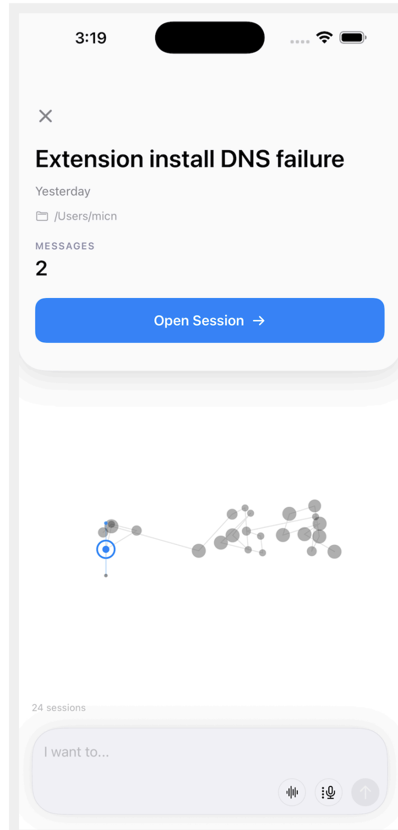
Prior attempts to deliver this application via claude-code and goose fell short of complete and usable results without further human instruction.

4 https://github.com/michaelneale/swifty-diff

Case Study: Mobile App Client in iOS

The goose agent has a desktop component which runs a server (daemon) which can be accessed remotely. In this case the g3 project was used to initiate and create a client iOS application[5] based on the API specifications and requirements for chat based interactions.

Similar to the diff application, this had follow-on human-in-the-loop interactive sessions (in this case far more) to complete the features, and move the application to a reviewable state. One of the weaknesses that required more interaction (instead of driving from requirements only) was that computer-controlling with iOS emulators are currently lacking. Web application automation and other desktop app automation is richer, and further research is needed to assist agents in controlling emulator environments to allow the coach to evaluate a rich mobile application.

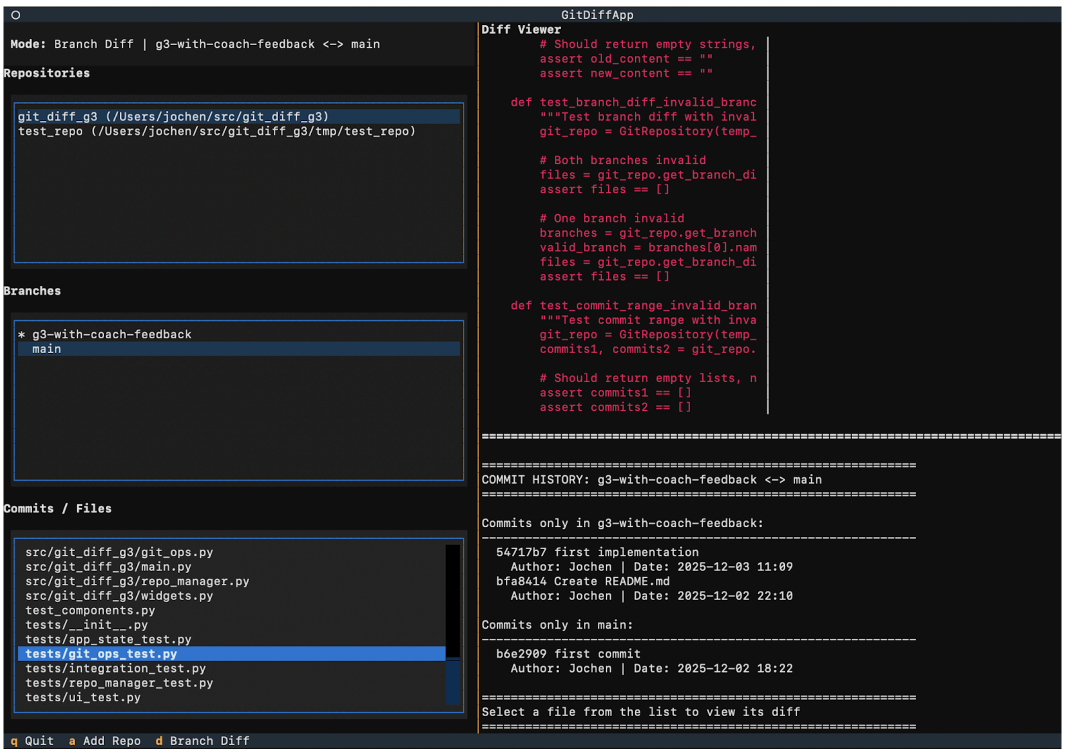


[5] https://github.com/dhanji/goose-ios

Case Study: Git Repo Explorer with Branch Diff Viewing

This was an attempt to compare the performance of a selection of leading coding platforms by asking for the implementation of a relatively simple application. The application was non-trivial, but simple and achievable enough that each platform had a fair chance of success. A terminal UI (TUI) was specified for ease of testing across the different platforms. (g3 was elsewhere tested with web, swift, MacOS, and mobile applications).

For consistency, we attempted to test the platforms with the same LLM - Claude-sonnet-4-5 with thinking mode. This was not possible on all platforms (see table below).



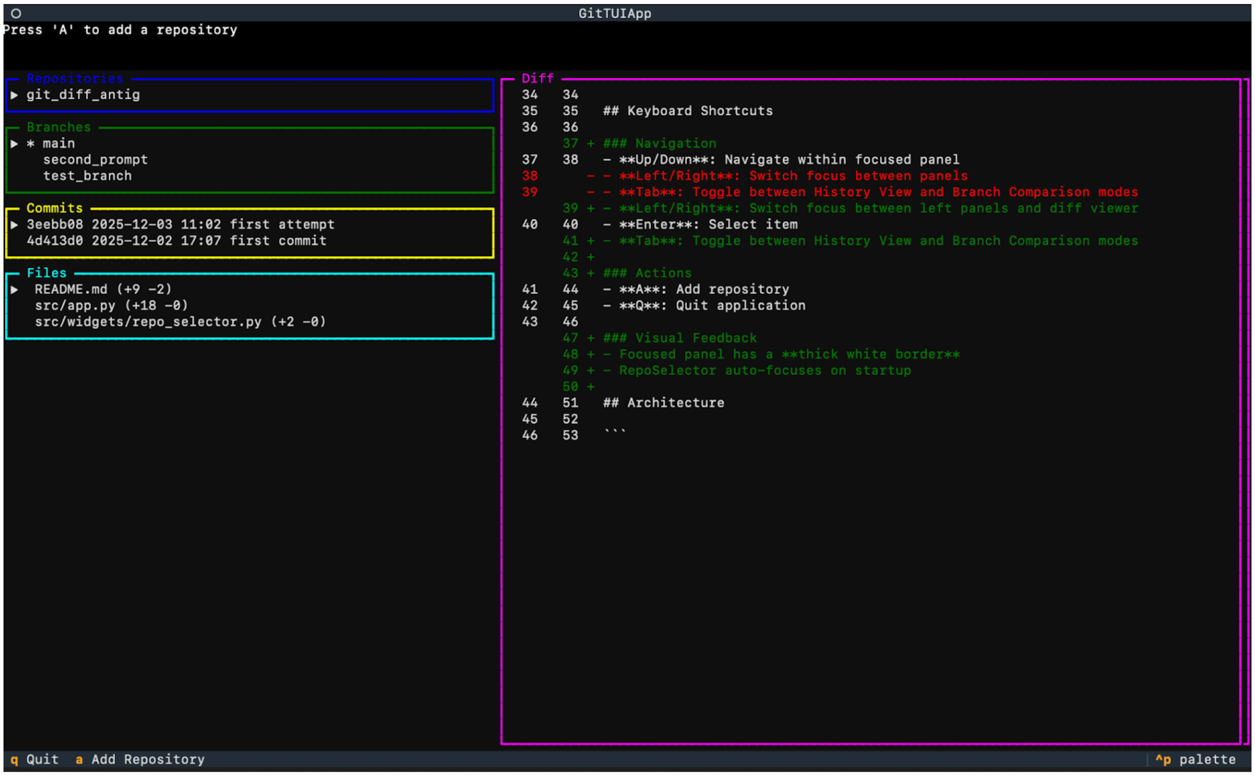
g3's implementation, note that it has the side-by-side diff, can do branch diffs, shows an attempt at commit tree.

In a one-shot scenario, no additional user input was given to the tool beyond the requirements. All platforms generated plausible looking code. After manual testing, we attempted to prompt the tool to fix the bugs or run the application, with mixed results. These attempts are mentioned in “implementation passes”. For g3 in autonomous mode (coach/player) it prompted itself, requiring no intervention. Only after a full run did we attempt to use the tool.

Note that the aim of running g3 in coach+player mode is to achieve autonomous coding. Thus emphasis is on one-shot implementation with no, or absolutely minimal user interaction, and running time is very much a secondary consideration.

For none of the tests did we spend extensive time trying to debug a crashing application. We attempted to get the platform to self-diagnose if possible, and where that didn't work, gave a stack trace or explained the problem. That was only attempted a maximum of 3 times before giving up.

	g3	Goose	Antigravity	OpenHands	VSCode Codex	Cursor Pro
Completeness (/5) final outcome	5/5 Meets all requirements. No crashes.	4.5/5 Very Functional, occasional crashes.	3/5 Initially crashed at startup. With prompting, it fixed that, but still crashes occasionally.	2/5 Incomplete, won't load branches.	1/5 Crashes on start. Couldn't fix it after some prompting.	1.5/5 Unable to load repo. After prompting it attempts to, but crashes.
Model	Claude-sonnet-4-5 with thinking	Claude-sonnet-4-5	Claude-sonnet-4-5 with thinking	Claude 3.5 Sonnet	GPT-5.1-Codex-Max	Claude-sonnet-4-5 with thinking
Observations about the implementation	Meets all the requirements, (visually not the most attractive).	Meets all the requirements, but has one or two bugs that cause it to crash occasionally (visually not the most attractive)	Has good UI, shows commit history separately. No side-by-side diff. No focus cursor. No branch diff. Occasional crashes.	Decent UI, but won't load branches.	Not working at all. A bunch of code has 'pass' in it. Looks poorly done.	UI suggests it might show the things we asked for, but it doesn't work.
Observation about the process & platform.	Ran in autonomous mode with no user interaction. Took a long time to run. Unpolished UI.	Ran the fastest. No user interaction. Good level of user feedback.	Very good user feedback in the UI, also fast. Good prompts to confirm prompt refinement and permissions. Great at controlling the app via keystrokes etc.. (needed explicit prompting)	Good level of user feedback. I couldn't upgrade the model. The test of the app kept hanging (no good support for killing process)	Decent level of user feedback in the UI.	Good user feedback in the UI.
Implementation passes	5 (autonomous)	1	2 (manual prompts to fix things)	2 (manual prompts to fix things)	2 (futile efforts to get it to test the app)	3 (manual prompts to fix things)
Automatically checked and verified test coverage	yes	yes	yes	yes	no	yes
LOC (incl tests)	1.8k	1k	1.4k	1.5k	1k	1.8k



Implementation via Antigravity. Note no side-by-side diff, can't do branch diffs, but has commit listing.

Time to completion varied considerably. Some runs involved unattended prompts that were blocked for an unknown amount of time, so we don't have an accurate time breakdown. Goose was the fastest at around 7 minutes, and g3 took the longest, around 3 hours. Given that g3 is a multi-pass approach, this is expected. (also, g3 is non-production code, optimizations have not been attempted. It is particularly slow in exploring the codebase, favouring token savings over aggressive code dumping and analysis). The other apps completed the task approximately between 20 minutes and an hour.

In an ablation study with g3, we withheld coach feedback. The player went 4 rounds of implementations with missing feedback. On each iteration it spontaneously found things to improve, however the final implementation was non-functional (it didn't prompt for a branch to read, and with explicit prompting to fix that, it still wouldn't load the git repo). The final outcome was on par with the OpenHands generated application (i.e. plausible code was written, tests were written, claimed to have implemented and tested everything, but was basically not functioning).

Noted chain-of-thought comments in the g3 logs when we did have coach feedback: "Let me check if the commit tree display is actually tested in the workflow test." And "Let me do a more thorough check of whether the implementation actually works as required. Let me verify the test for multi-file commit workflow more carefully."

Project spec given to tool/platform as requirements:

```
Git Repository TUI Viewer - Project Specification

A Python terminal user interface (TUI) application for exploring and comparing Git repositories without
modifying the working directory state. The application provides a multi-pane interface for navigating repository
history and viewing changes. Use the https://github.com/Textualize/textual library for UI & rendering, you can
use git libraries such as (GitPython). Navigation should be exclusively with cursor keys and tab.

The application maintains a list of user-added Git repositories with the ability to switch between them via a
selection panel. For the actively selected repository, users can browse all branches (including worktree
branches) and their commit history in a navigable list view. Selecting a commit displays its changed files, and
selecting a file renders a side-by-side diff view on the right-hand side of the screen. All repository
inspection is performed using Git's object database directly, never checking out branches or modifying the
working tree.

Branch diff mode:
Users can select two branches to compare, which displays a list of files that differ between them. Navigating
this file list shows side-by-side diffs of each file's content between the two branch heads. The interface
should use a Python TUI library (such as Textual) with keyboard navigation throughout, maintaining a consistent
layout with navigation/selection on the left and diff content on the right.
Also show the commit history dependency tree between both branches underneath the side-by-side diff view.

The interface uses a consistent split-pane design: left side contains fixed navigation elements (repo selector,
branch list, commit list, file list), the right side displays diff content (side-by-side). A mode indicator
shows whether the user is in single-branch history view or branch comparison mode. All navigation uses cursor
keys with clear visual focus indicators.

Important for evaluation and testing:

RUN the application yourself to make sure it works. Carefully think about how you will ensure it works as
intended. Send keystrokes to the app to make sure you can navigate and load repos.

Make sure to write high quality code, and have great test coverage for your code, I will run a code coverage
tool.

Guideline for code design:
- Functions and methods should be short - at most 60 lines, ideally well under 40.
- Classes should be modular and composable. They must have between 2 and 20 methods.
- Do not write deeply nested (above 6 levels deep) ‘if’, ‘match’ or ‘case’ statements, rather refactor into
separate logical sections or functions.
- Code should be written such that it is maintainable and testable.
- For Python code write *ALL* test code into a top level ‘tests’ directory.
- Each non-trivial function should have test coverage. DO NOT WRITE TESTS FOR INDIVIDUAL FUNCTIONS / METHODS /
CLASSES unless they are large and important. Write tests that tests multiple functions or components at once, do
only minimal mocking, and *DON'T ADD TRIVIAL TESTS THAT ONLY CHECK PARAMETER-PASSING AND THE REST IS MOCKED
OUT*.
- Write tests in separate files, where the filename should match the main implementation and adding a “_test”
suffix.

General guidelines for code design:
- Keep the code as simple as possible, with few if any external dependencies.
- Don't repeat functionality across the code, unless there is very good reason.
- Keep each function/method/class simple, avoid unnecessary complexity.
- Implement features that you actually need, not for future potential expansion. Do not over-engineer.
```

(The code quality part is probably superfluous given that the system prompt of various platforms will say similar things. It was included here, however, in an attempt to standardize between platforms).

Implications and Recommendations

Dialectical autocoding in the "coach/player" paradigm if viewed as a pattern can be replicated today with almost any agent on the market. This can be used at the very least as a step improvement in aiding agents converging on an end-to-end working solution with less interruptions, and perhaps if implemented more natively provide a way to scale parallel development allowing the human-in-the-loop to maintain deeper focus on tasks where needed. We further believe that this can overcome the nights and weekends problem, with multiple competing paths in an experiment cluster taken simultaneously rather than having to restrict by human resource availability.

Not only does this dramatically increase the efficiency of ai coding tools of today, but with far fewer human-in-the-loop turns required, opens up significant possibilities for non-technical people to develop competent software.

We propose to add adversarial cooperation as an enhancement to goose in the near future.

References and Notes

1. https://github.com/dhanji/g3
2. https://www.tbench.ai/leaderboard/terminal-bench/2.0
3. https://dl.acm.org/doi/10.1145/1281700.1281702
4. https://github.com/michaelneale/swifty-diff
5. https://github.com/dhanji/goose-ios