Losing Control and Getting It Back

A field report on using LLMs with a little-known language

Chris Armstrong – Founding Principal Engineer

Sydney AI Engineering Meetup · 4 March 2026

The Journey

Three OCaml projects, each using LLMs differently.

smaws

LLM-assisted

AWS SDK for OCaml.
Code generation using Smithy interface definitions.
Presented at ICFP 2025 OCaml Workshop.

ocgtk

Vibe-coded

GTK4 binding generator.
~20k line code generator produces ~140k of bindings.

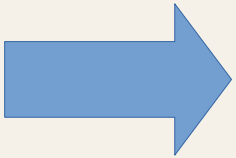
alan

Guardrails-first

Community website built on an OCaml-native stack.
Structured planning and code guidelines with new libraries.

Why OCaml?

- Fast builds
- Fast executables
- Statically typed, sound type system
- Compact & concise
- Pragmatic programming paradigms (Functional, OOP, & Imperative)
- Relatively simple language with standard style (no “dialects”)
- Mature (30+ years)



Ideal language for use with LLMs

ACT 1 - smaws

“Lets accelerate some work”

Using Claude Sonnet 4 to generate HTTP protocol compliance tests from Smithy JSON specs

The Good

Scaffolded the unit test code generator to create HTTP “request tests” & refactored the framework to implement a mock HTTP client
(all in time for a conference presentation in 2 weeks!)

The Gap

Code generators are an abstract scenario for an LLM. The context required constant reseeding and pointers to help it understand what it was working with, especially understanding OCaml’s AST.

The model didn’t understand what it was trying to achieve and would frequently “forget”. Without context seeding, it was necessary to remind it where to look, when to build and test the code, and how to write code generation.

ACT 2 - ocgtk

"When the casino is free"

AU$250 of free credits. 2 weeks to usedthem, and only through Claude Code Web.
A interesting side project I had never found time for.

The project

Port lablgtk3 → GTK4 bindings for OCaml

No complete GTK4 bindings existed. Good reference code to work from. .

The conditions

Busy with day job, low headspace

Cursory glances at GitHub PRs before merging

Re-bootstrapping the OCaml toolchain every session (not in default container image)

None of my own money was on the table, so what did it matter?

The cost of not paying attention

I thought I was doing agent coding the right thing by writing a plan

- port the “core” to GTK4
- slowly enable code generation in later stages
- create small set of examples (e.g. forms, calculators) for high-level validation

But this isn’t what happened

Sonnet 4.5 kept “pushing off” the requirement to build a code generator, saying it wasn’t ready yet. Without forcing it to re-plan around code generation (and allowing it to turn off tests), it kept pushing ahead with porting existing code.

The “code generation” was nothing more than some enumeration bindings (with LLM-generated interface definitions)

Plan B: gir_gen

Discarded the existing plan and replaced with a new plan to:

- Exclusively use programmatic code generation (“gir_gen”) – only retain small core of “runtime” code while removing LLM-generated code
- Switch off any failing tests as we go – focus on code generation coverage

Three Layers

Layer 0
`c_stub.ml`

C stub generation: bindings to GTK’s C interface

Layer 1
`ml_interface.ml`

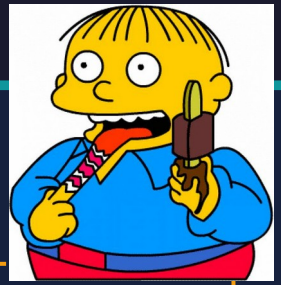
OCaml interface types and function definitions that bind to C

Layer 2
`class_gen.ml`

Class wrappers recreating the GTK class hierarchy

Keep the party going: multi-agent development

*The key to multiple agents is non-determinism – executor and reviewer **run at least once per stage***



Coordinator

`variable`

Planner

`Opus / Sonnet`

Called once. Produces a set of tasks from the goal (JSON). Update plan as gaps are found.

Loops per task (and on failed reviews)

Executor

`variable model`

Implements ONE task.

Fresh context each time.

Must run build + tests before reporting done

Output: summary of execution.

Reviewer

`Sonnet/Opus (different from executor)`

Sees only goal + result + changes.

Runs build + tests independently.

Output: PASS / FAIL

FAIL

Why coordinate separate agents?

- **Performance**: run longer without supervision
- **Stability/Reliability**: Fresh context each time => less model degradation and “forgetting”

but primarily **Cost:**

output tokens are typically 3x - 4x cost of input tokens

token cost is roughly correlated with power/stability/reliability

less powerful models for execution
more powerful models for review & planning

The seduction of velocity

The Ralph loop helped to build the functionality with less supervision, but at a cost

~20k
lines: LLM-assisted code generator

~140k
lines: generated GTK4 bindings

65
memory-related bug fix commits

0
comprehensive tests (initially)

An agentic loop just produces unreadable code faster!

Code generation functions hundreds of lines long · Match statements nested 5–6 levels deep · Filtering logic duplicated across layers with subtly different abstractions · XML parser with fragile tag-matching that silently skipped entire sections · Naming conventions incoherent between layers · C stubs missing proper memory management

Coding without supervision or guardrails

> Sonnet couldn't implement changes without regressions. Without test coverage, every modification was a coin flip. I had to stop all feature work and build test suites before I could safely refactor anything.

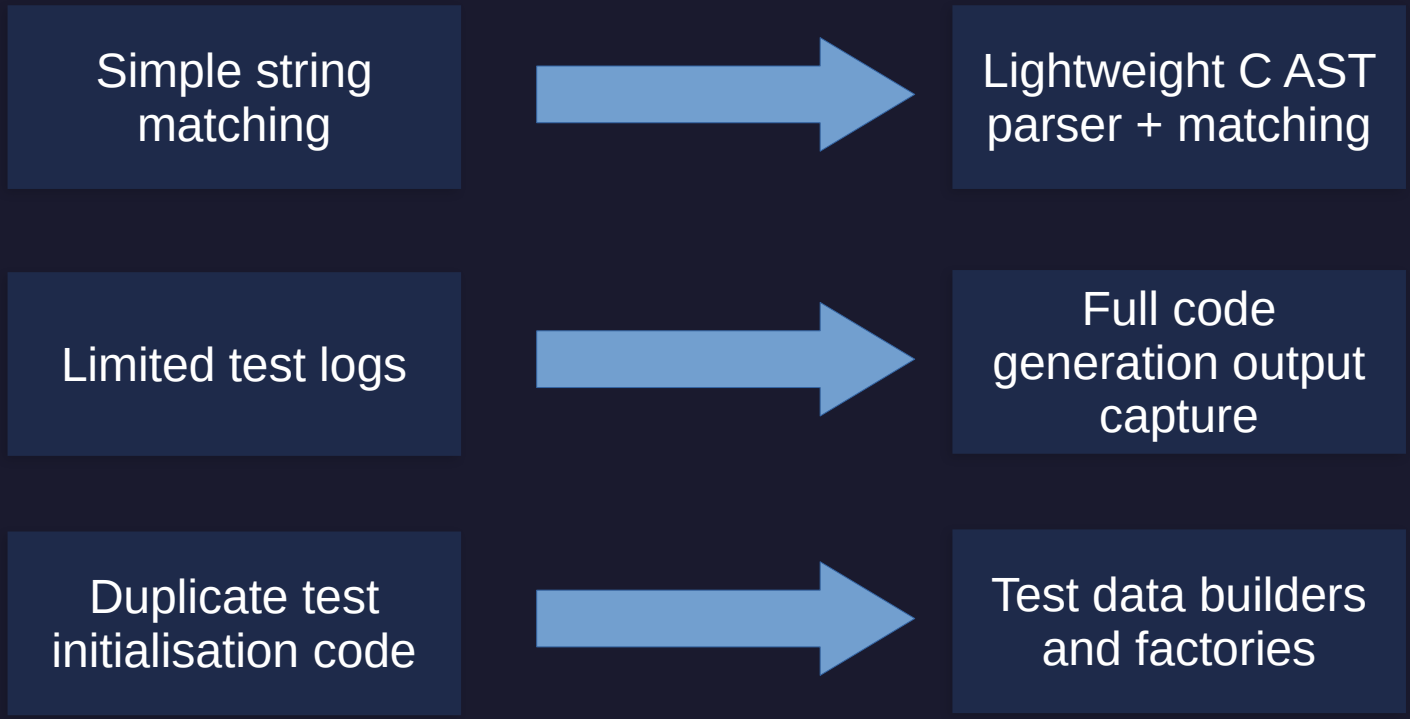
Creating test coverage: C stubs first

The code generator had three layers, each tested independently. C stub generation was where most time was spent as it had the most complex semantics.

> *LLMs are less good at following instructions than they are at picking up the vibes from the sea of text given to them.*

Extending and improving tests

Tests need to be refined just like your code – LLMs will take shortcuts here the most & make future iteration harder



Refactoring with guidelines

> Coding guidelines that are linked into agent prompts ensure new code meets our ideal…
>
> …but old code must be brought up to standard somehow

Refactoring the mess

The planner was instructed to find code violating the guidelines and write a refactoring plan.

Improved test coverage provided confidence that code was not regressing as it was refactored.

> *This process is not like feature implementation: it took multiple iterations with explicit instructions to the planner to get the desired test and code quality.*

Confusing the LLM

Working at language boundaries with ill-defined semantics

- **Code generation complexity from the GObject IDL:**
  - GIR has several C memory management conventions
  - And lots of parameter passing semantics (in/out/inout)
  - …and in combination interact differently with each type (integer, strings, records, arrays, etc)
- **Lack of examples**:
  - e..g. determining how to implement C ↔ OCaml array passing was particularly painful – multiple attempts as there was no reference code
- **Model training incomplete**:
  - Model defaults to looser OCaml 4 pointer semantics while we're using OCaml 5

ACT 3 - alan

Starting the right way

Incorporating the lessons from vibe coding and agentic loops

- **Requirements Document**: source of truth for any implementation plans
- **Coding guidelines** adapted from ocgtk
- **Agent Command Loop** with executor/reviewer/planner agents
  - **Executor agents** run build and test commands before finishing
  - **Reviewer agents** strictly apply coding guidelines
- **Implementation plans**
  - **“phases/stages/parts”** that implement a slice of functionality
  - **Test creation** is incorporated into the plan
- **End-of-phase implementation review** of code and executable outputs, with “re-plan” for fixes (Phase 1 → Phase 1A → ...)

Model tiering: approximate results

Planner / Reviewer	Executor	Verdict
Sonnet	`M2.5`	Slowest, least reliable
Sonnet	`Kimi K2.5, Haiku`	Struggled with applying OCaml idioms
Sonnet	`GLM-5`	Much faster, reasonably on track
Sonnet	`Sonnet`	Better, still nests matches without instruction
Opus	`Opus`	Best quality, expensive

Once surrounding code was refactored into idiomatic patterns, even Minimax M2.5 started replicating correct structures. The model learns from your codebase, not its training data.

Why bother with tiering? A Claude Max subscription is convenient, but for less money you can combine Opus/Sonnet for planning and review with frontier models like GLM/Kimi for execution — running more projects in parallel without hitting usage limits.

Use the right model for planning

Same task (Phase 2 implementation plan). Same codebase. Different planning models.

Opus 4.5 — **1,900 lines**

- ✔️ Typed database columns
- ✔️ Hexagonal architecture with correctly identified ports
- ✔️ 100+ tests: 58 E2E + per-stage unit tests (strenuous!)
- ✔️ Explicit dependency graph + ordering
- ✔️ References existing commits, adds fix stages
- ✔️ Surfaces coding guidelines compliance needs

MiniMax M2.5 — **640 lines**

- ✘ 3–5 tests per stage, no E2E
- ✘ No dependency graph or ordering
- ✘ Features deferred as placeholder
- ✘ Doesn’t engage with existing codebase
- ✘ Ignores existing commits & coding guidelines

Cheap executors can follow a good plan, but struggle to write one with enough detail

Your codebase is the prompt

Written guidelines cover far more than code style:

- **Nesting & control flow** — max 2 levels, bind operators
- **Error handling** — Result types, never exceptions for control flow
- **Pattern matching** — exhaustive cases, no hidden catch-alls
- **Ban "unsafe" functions** — throw exceptions (List.hd, Option.get, etc.)
- **Type safety** — phantom types for IDs, no stringly-typed errors
- **Module boundaries** — .mli files required, layered architecture
- **Abstractions** — when to extract ADTs, phantom & abstract types, etc.
- **Code reuse** — DRY, ensure utilities are shared and not duplicated
- **Naming** — full descriptive names, positive booleans
- **Logging** — per-module logger sources, no Printf
- **Database** — INSERT completeness, tuple nesting, upserts
- **Test patterns** — AST-based validation, full test assertions

Coding guidelines split over multiple files used in execution and review.

Each one written in response to a specific failure mode the models kept hitting.

Ensure model alignment and reduction of pathological behaviours.

Evolving guardrails as you spot issues

alan — applying lessons learned, then refining them in-flight

Day 1 — **10 guideline files seeded from ocgtk lessons**

Nesting, errors, naming, pattern matching, type safety, tests, abstractions

Day 6 — **+logging guidelines, +221 lines on avoiding nested pattern matching**

Show the model how to use monadic bind operators and utility functions instead of nested match statements

Day 7 — **+database guidelines (3 commits, +543 lines)**

How to use the database library, INSERT completeness, upsert patterns to avoid the model getting tripped up

Day 9 — **Agent config overhauled — reviewer switched to GPT-5.3-codex then Sonnet**

Cross-vendor: Sonnet plans, M2.5 executes, GPT reviews

Day 10 — **+ Planner verdict**

Planer rejects structurally unworkable stages, not just buggy implementations

Day 12 — **Split coding guidelines**

Define *mandatory* guidelines (typically style, readability, acceptable functions) and *advisory* guidelines (specific implementation area like database, mlx, etc.)

Code issues were spotted both manually and through post-implementation Sonnet/Opus review.

Each time I would plan and execute a refactor and update the guidelines.

Lessons learned

1. **Plan upfront, refine as you go.**

   Write an implementation plan before any feature or refactoring. Update it as gaps emerge: add stages to address issues rather than hacking around them.

2. **Build test suites to enable safe iteration.**

   Whether you write them by hand or with the LLM, comprehensive tests are what let the model iterate safely. Without coverage, every modification can regress your code.

3. **Set up a plan-execute-review loop from the start.**

   Even a simple setup lets you leave agents running longer and catch errors before they compound. The executor and reviewer should always run the build and tests themselves *and make sure they are clean*.

4. **Be disciplined about refactoring.**

   If a stage produces suboptimal code, don't move on: instead, implement an ad-hoc refactor stage. Plan it, run it through the same loop, and keep the codebase clean for the next stage.

Lessons learned

5 You need comprehensive coding guidelines upfront.

With niche (and especially FP languages), models default to imperative patterns in functional code, resulting in nested pattern matching, stringly typed errors, and more bespoke utility functions.

Written guidelines covering nesting, error handling, banned functions, and type safety are essential *for all models*.

6 Review, refine, refactor.

Review your code yourself and perform post-implementation reviews with a larger model, as the end-of-stage reviews won't always catch poor abstractions. **Refine** your coding guidelines when you spot patterns. **Refactor** in planned stages between features to improve quality.

7 Frontier models can execute, but not plan (and should avoid for review).

Kimi 2.5, GLM-4.7/5 work well for execution inside a good plan, but lack the thoroughness for planning. Use SOTA large models (Sonnet/Opus/Codex/etc.) for planning and review

(review should always be a different model that execution, and ideally a larger one with more "depth of thought")

Chris Armstrong

Github: chris-armstrong

Blog: https://www.chrisarmstrong.dev

LinkedIn: linkedin.com/in/christopher-armstrong