Failure Driven Development

Paul Tarvydas

Failure Driven Development

Software Development Habits Haven't Caught Up to Cheap Compute

Paul Tarvydas

This book is available at https://leanpub.com/failuredrivendevelopment

This version was published on 2026-05-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Paul Tarvydas

Contents

Introduction 1

Overview 2
- Who Is This Book For? 3
- What You Can Do Differently 4
- The Design Problem 4
- Zero Defects In The Field 5

Are We Really Better Than Sketchpad? 6

Concurrency and Parallelism 7
- Concurrency Is Not Parallelism 7

Code Perfection vs. Design Perfection 8

The Failure Driven Development Workflow - Top Down and Bottom Up 9

The Synchronous Straightjacket - Why Today's Programming Cannot Solve Tomorrow's Problems 10
- Programming Is a Subset of What a CPU Can Do 10
- The Economics of the Mistake 11
- Symptoms: Thread Safety and Concurrency 11
- Symptoms: Callback Hell 12
- The Internet Is Not Synchronous 12
- The Dark Matter Analogy 13
- The Notation Trap 13
- The Workflow, Not the Language 14
- The 20th Century as Stepping Stone 14
- What Needs to Change 15

Changing The Software Development Workflow . . . 17

Multiple Notations in a Single Project . . . 18

Making Erasure Cheap . . . 19

Black Boxes, Not Functions . . . 20

Little Languages and Small Tools . . . 21

T2T: Text To Text Transmogrification . . . 22

Bench Testing and Component Palettes . . . 23
Testing Examples . . . 23

Knowing When to Commit . . . 24

FDD in Practice . . . 25

Example FDD Workflow - The Five Whys Example . . . 26
Getting the LLM Running . . . 27
Wrapping the LLM as a Part . . . 29
What I Learned From This Test Jig . . . 33
Build a Stub Test Jig (No API Calls) . . . 33
Count To Five . . . 35
Rethink and Retrench . . . 35
Continue Count To Five . . . 36
New Insights, Pass 2 . . . 39
Different Ways to Probe . . . 44
Internally Rephrasing Answers as Questions . . . 47
New Part For Rephrasing Answers . . . 49
Make the Debug Output More Concise . . . 60
Conclusion . . . 64

Parts Based Programming . . . 66
Suggested Starting Rules . . . 66
Resources . . . 66
Everything is a Black Box Part . . . 66

Drafting Rules For Node-and-Wire Diagrammatic Languages 66

Building Multi-Language Tools 68

@make 68

Appendix A - How We Got Here 69

Historical Choices 70

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Overview

During development, programs tend to fail.

After a program is debugged, it stops failing. You ship it and move on to develop another program – which, also, mostly fails as you debug it.

The name “waterfall development” is given to the idea of being so sure that you know about some aspect of a problem space – or about the whole problem space – that you can hard-code it as software in a program.

The actual *development* workflow, though, does not resemble “waterfall development”. You learn new things about the problem space as you develop and debug.

The ideal workflow, then, is to iterate and to change your mind as you gain new insights about a problem. If you hard-code parts of the problem too early, and you work hard at it, you feel reluctant to change your mind and rewrite the hard-coded software, when you learn new things about the problem.

Thinking and learning are hard. Code should be cheap.

3GL programming languages, like Python, Rust, etc., encourage early hard-coding. Type-checking encourages early hard-coding, too. When you learn something new about the problem, you *should* revamp the software and revamp the type system, but you tend not to, since you put in a lot of hard work to get to where you already are.

This situation is not an ideal development workflow. Waterfall development works when you know *everything* about a problem space, but it doesn’t work so well while you’re still learning about the problem space. Waterfall is what you want to do as a Production Engineer, but not as a Design Engineer. You can apply waterfall thinking when the design has stabilized. Applying waterfall in the early stages leads to delays and poor designs.

Fred Brooks, in his famous book, the *Mythical Man-Month*, said that you fail, then you fail, and only then do you succeed. That’s a failure rate of 67%.

The modern software development process is much like Waterfall development. You start out by assuming that you know what you want to build then use

modern programming languages to methodically build a single artifact. Code is immediately calcified by laborious attempts at developing consistent type systems and premature optimization. FDD, though, assumes that we don't know enough about the problem domain at the outset. We iteratively explore and poke at aspects of the problem domain in a manner similar to early REPL based languages and RAD (Rapid Application Development). The FDD workflow strives to make it easy to learn about aspects of the problem domain and to change your mind. You can erase all code easily and begin from scratch, retaining newly acquired knowledge without being discouraged to erase bits of code due to having invested too much time in polishing the stuff you already wrote, e.g. the type system, the code workarounds, code that you've spent too much time making efficient. When your code no longer fits with all aspects of the problem domain, you must be able to throw out the code easily. FDD recognizes that this sort of iterative process must be used and strives to remove barriers to starting all over again. Furthermore, our belief in "computation" drives us towards assuming that everything can be described as functions – 1-in and 1-out functions don't easily describe most problem domains, except the calculator (e.g. ballistics) domain, and, the sequential call/block/return control flow imposed by the functional paradigm makes it difficult to describe non-sequential, asynchronous, parallel processes that have become more common in modern problem domains, like distributed computing, internet, robotics, etc. We need to be able to use components with multiple inputs and outputs that can be easily unplugged from a design in a black-box like manner. Pin for pin compatible. The functional paradigm imposes severe restrictions on this approach and calls it "referential transparency", but, this is not the only way to achieve black-box pluggability. The fact that we *can* twist a problem to fit the functional paradigm does not mean that we *should* use the functional paradigm to create solutions for every problem.

Who Is This Book For?

Software practitioners who need to develop solutions to problems using current tools and programming languages. Those who are more interested in robust designs needed by customers rather than just robust code as envisioned by techies who aren't deeply connected with users and don't have enough experience with shipping what customers want. A goal should be to have zero defects in the field and to not need to rely on Continuous Delivery that

uses customer complaints and customer bug reports and customer design improvement suggestions as a way to (slowly) iterate the design.

What You Can Do Differently

1. You should be able to build code that helps you explore a problem space and to be encouraged to go back to the drawing board every time you learn something new.
2. You should be able to do this using existing languages and tools and workflows. Nothing drastically new, except a fresh way to approach the design problem.
3. You should be able to figure out when to draw the line and to choose a design, even if not perfect, to drill down onto, perfect and ship.
4. You should be able to "bench test" designs quickly and easily and weed out design and coding problems early, before committing to a specific product design.
5. You should be able to build up a palette of useful software components that you can plug into and out of design iterations.
6. You should be able to build little languages and little networks of black boxes in hours instead of months.
7. You should be able to break the habit of over-optimizing too early, to build simple black boxes that help you speed up development workflow turn around.
8. You should stop equating the concept of "programming language" and "DSL" with heavy weight Turing complete behemoth general purpose programming languages and use (and develop) small tools that help you speed up development by allowing you to choose and use appropriate paradigms for thinking about different aspects of your problem.

The Design Problem

We have developed methods to automatically check consistency and correctness of code that we write.

We have not spent as much effort in developing methods to gather requirements from customers nor on checking that our code corresponds to customers' needs instead of just checking that code corresponds to programmers' ideas of what customers need.

Customers, when asked, do not think in enough detail to communicate to programmers what they need, in a consistent way. Programmers want code rigor, customers tend not to be interested in code rigor. Customers focus on their own problems and not on programmers' problems. Often customers' focus on their own problems is not sufficiently rigorous to be translated into computer programs, so programmers "invent" semantic bridges that satisfy the need for self-consistent code, while usually missing what customers actually need.

Zero Defects In The Field

The goal should be to deliver code to customers. Code that never needs to be updated. Code that has no bugs in the code nor in the design.

The delivered product doesn't need to be "perfect". It only needs to improve some aspect of customers' workflow and remains stable - doesn't crash and doesn't change.

Customers don't want a plethora of features instead of simple, robust tools. For example, my hand-held calculator never crashed. My mother's Tetris game crashes often, even though it runs atop a 55,000,000 LOC operating system and has zillions of features.

What does "zero defects in the field" actually require – and why it starts with design, not code review.

The practitioner's frustration: when technically excellent code repeatedly fails to solve real customer problems.

Are We Really Better Than Sketchpad?

Are We Really Better Than Sketchpad?

Concurrency and Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Concurrency Is Not Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Analogy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Multi-Threading Fakes Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

True Parallelism Requires Dedicated CPUs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

PBP — Parts-Based Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Code Perfection vs. Design Perfection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

The Failure Driven Development Workflow - Top Down and Bottom Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

The Synchronous Straightjacket - Why Today's Programming Cannot Solve Tomorrow's Problems

What we call "programming" today is not a neutral description of how computers work. It is a historically contingent choice – made in the early days of computing – to model machine behaviour using mathematical notation designed for paper. That choice was reasonable for the problem domains of the time: sequential, synchronous, single-threaded. It is not reasonable for the dominant problem domains of today, which are asynchronous and non-sequential. The complications we treat as inherent to software – thread safety, callback hell, cache coherency, concurrency primitives – are not inherent. They are symptoms of applying the wrong model to the wrong domain. We cannot fix this by inventing more programming languages built on the same base paradigm. We need a different paradigm.

Programming Is a Subset of What a CPU Can Do

A CPU integrated circuit processes instructions in a fetch-decode-execute loop, manages registers, and communicates through busses and pins. It is, at bottom, an asynchronous piece of hardware embedded in an asynchronous world.

What we call "programming" is not a faithful notation for this. It is a layer of abstraction built on the assumption that computation is sequential, synchronous, and single-threaded. That assumption was chosen, not discovered. It was chosen by people who believed that mathematical notation – developed over centuries for use on paper – was the best available tool for generating machine code.

The mismatch between the notation and the hardware is real, but it remained invisible for decades because the problems programmers were asked to solve were themselves sequential and synchronous. The notation and the

problem domain were mismatched with the hardware, but they were matched with each other. That coincidence shielded us from the consequences.

The Economics of the Mistake

In the early days of computing, hardware was expensive and human time was cheap. It was rational to spend engineering effort economising on CPU cycles and memory bytes. It was rational to squeeze every problem into a form that a single processor could address sequentially, because that single processor represented a significant capital investment.

The economic situation has since inverted. CPUs are now cheap enough to be disposable. Memory is measured in gigabytes and costs almost nothing. Human development time, by contrast, is the most expensive resource in any software project.

Despite this inversion, the field has not changed its methodology. We continue to use techniques invented to conserve expensive hardware even when hardware is cheap and the techniques themselves are expensive to apply correctly. We call this conservatism "progress".

Symptoms: Thread Safety and Concurrency

The clearest symptom of this mismatch is the concept of thread safety. What we call "concurrency" is not a natural property of computing. It is a technique called time-sharing, invented to extract maximum utilisation from a single expensive processor by switching rapidly between tasks and creating the illusion of simultaneous execution.

Time-sharing requires that all the tasks sharing the processor agree on how to access shared memory. When they do not agree, or when the agreement breaks down, the result is a race condition. Enormous intellectual effort has been invested in preventing and detecting race conditions: mutexes, semaphores, monitors, lock-free data structures, memory models, and formal verification techniques. This entire edifice exists because we are running multiple logically independent tasks on a single processor that was designed for sequential execution.

The hardware alternative – multiple processors, each handling its own task, communicating through messages rather than shared memory – was always

technically cleaner. Hardware engineers imagined it from the beginning. It was economically inaccessible then. It is economically trivial now. We continue to use the complicated approach not because it is better but because it is the approach we already know.

Symptoms: Callback Hell

The rise of asynchronous programming on the internet produced a phenomenon programmers called "callback hell". A program that needed to perform several operations in sequence – each of which might complete at an unpredictable time – had to nest its logic inside a cascade of callback functions. The resulting code was difficult to read, difficult to reason about, and difficult to maintain.

Callback hell was not a bug in JavaScript or in any specific platform. It was a symptom of forcing an asynchronous problem into a synchronous notation. The notation assumes that execution proceeds in a straight line, one step after another, with each step completing before the next begins. An event-driven, network-connected application does not proceed this way. The mismatch between the model and the reality produced the pathology.

The field's response was to paper over the symptom rather than address the cause. Promises, `.then()` chains, `async/await`, and reactive frameworks are all sophisticated ways of pretending, within a synchronous notation, that asynchrony can be managed. They reduce the visible ugliness of callback hell. They do not change the underlying model.

The Internet Is Not Synchronous

The internet is the largest and most consequential software system ever built. It is also, in its fundamental nature, asynchronous and non-sequential. Packets travel through unpredictable routes and arrive at unpredictable times. Services fail and recover independently. Users interact with systems at uncoordinated moments from arbitrary locations.

Solving internet-scale problems with synchronous, shared-memory techniques is not a matter of applying the right tools carefully enough. It is a category error. "Synchronous" and "asynchronous" are not points on a

spectrum. They are opposites. You cannot solve an asynchronous problem with a synchronous model any more than you can solve a three-dimensional problem with a two-dimensional notation. You can solve only those instances of the problem that happen to reduce to two dimensions. The rest remain unsolved.

The complications that have accumulated around distributed systems programming – consistency models, consensus protocols, the CAP theorem, saga patterns, eventual consistency – are not discoveries about the nature of distributed computation. They are the accumulated cost of continuing to apply synchronous thinking to an asynchronous domain.

The Dark Matter Analogy

Astrophysicists know that the visible matter they can detect and measure accounts for roughly five percent of the universe. The remaining ninety-five percent is accounted for by placeholder terms: dark matter and dark energy. These terms do not describe anything understood. They describe the gap between what the current model can explain and what observation requires.

The synchronous programming paradigm stands in a similar relationship to the full space of computational problems. The problems it can address cleanly – sequential, deterministic, single-threaded – are a small fraction of the problems that computing is asked to solve. The rest are handled through workarounds, approximations, and accumulated complexity, or they are simply not solved. We do not always notice the unsolved remainder because the workarounds have become so familiar that they seem like the natural state of affairs.

Asynchronous problems that cannot be squeezed into synchronous solutions are not exotic edge cases. They are the majority of the problem space that the internet, robotics, embedded systems, and real-time applications represent.

The Notation Trap

Every programming language invented in the last several decades – C, Haskell, Rust, Clojure, Python, Go – is, at its base, a synchronous sequential notation.

The syntactic surface differs. The type systems differ. The memory management models differ. The underlying assumption does not differ. Execution proceeds in a line. Functions call other functions and wait for their return. State is global or passed explicitly. Time is not a first-class concern.

Adding concurrency features to these languages does not change this. It adds machinery for managing the illusion of simultaneity within a model that does not natively support it.

The belief that a sufficiently general-purpose programming language can address all problem domains is itself a product of this trap. It assumes that one notation can be adequate for all domains. This assumption is not self-evident. Mathematicians use different notations for different branches of mathematics. Engineers use different tools for different phases of a design. The demand that all software be written in one language, or in languages sharing one paradigm, is an historical artifact, not a logical necessity.

The Workflow, Not the Language

The unit of computation that needs to be Turing complete is not a programming language. It is a workflow.

A workflow is a composition of tools, each of which addresses a specific concern in the form most natural to that concern. Some steps in a workflow might be handled by a declarative pattern-matching notation. Others by an imperative script. Others by a diagram. The workflow as a whole can be Turing complete even if no individual step is.

This is how hardware design already works. A hardware engineer composes schematics, simulation tools, synthesis tools, and timing analysers into a workflow. No single tool in that workflow is expected to solve the whole problem. The composition solves the whole problem.

Software inherited the single-language assumption from the constraints of early computing environments – limited storage, limited tooling, the difficulty of integrating disparate notations. Those constraints no longer hold. The assumption persists only through habit and through the institutional weight of accumulated investment in existing tools.

The 20th Century as Stepping Stone

The techniques of 20th century computing were not mistakes. They were rational responses to the constraints of their time. Sequential, synchronous programming gave us reliable compilers, operating systems, databases, and a generation of programmers who could be trained systematically. That body of work is real and valuable.

What is a mistake is to treat those techniques as the destination rather than as a stepping stone. The 20th century gave us cheap CPUs. It gave us abundant memory. It gave us high-speed networks. It gave us a clear view of problem domains that are asynchronous and non-sequential in ways that were not visible when the original methodological choices were made.

The 21st century problem is to use what the 20th century built as a foundation for something different: a methodology suited to asynchronous composition, to multiple notations in a single workflow, and to the use of many cheap processors rather than the intensive exploitation of one expensive one.

Hardware engineers designed for this world from the beginning. A schematic does not have a stack. An integrated circuit does not block. Components in a hardware system communicate by sending signals, not by calling functions and waiting. The programming discipline that builds on 20th century techniques needs to learn something from the discipline that produced those techniques in the first place.

What Needs to Change

The change required is not a new programming language. It is a change in what programming is understood to be.

Programming needs to be understood as the composition of isolated components that communicate by message-passing, where the wiring between components is specified separately from the components themselves. Each component does not need to know what is connected to it. It sends outputs. It receives inputs. What connects those inputs and outputs to the rest of the system is a concern of the containing system, not of the component.

This is how hardware ICs work. It is not how software components work today. Software components are coupled to their callers through shared call

stacks, shared return addresses, and shared memory. That coupling is what makes thread safety hard, what makes callback hell appear, and what makes distributed systems so difficult to reason about.

Decomposing software into truly isolated, asynchronous components – components that do not share memory, do not call each other directly, and do not block – does not eliminate complexity. It moves complexity to a level where it belongs: the composition level. It makes the behaviour of individual components easy to understand and test. It makes the behaviour of a composed system easy to modify.

This is not a novel idea. It is the idea that hardware engineers have applied successfully for decades. The novelty is in recognising that software can be designed the same way, and that the reasons it has not been are historical rather than fundamental.

Changing The Software Development Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Multiple Notations in a Single Project

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Making Erasure Cheap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Black Boxes, Not Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Little Languages and Small Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

T2T: Text To Text Transmogrification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Bench Testing and Component Palettes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Testing Examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Knowing When to Commit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

FDD in Practice

TODO: This chapter covers:

- A practical cookbook tying together Chapters 3–7 into a workflow you can start using on Monday – with existing tools, languages, and IDEs
- The FDD starter workflow: step-by-step from "I have a new problem" to "I have a committed component in my palette"

  1. Frame the exploration: what aspect of the problem domain are you probing?
  2. Run the first experiment: smallest possible probe – no types, no tests, no optimization
  3. Capture what you learned: update your design knowledge before touching the code again
  4. Sketch the black box API: input events, output events, failure modes
  5. Bench test the API with scripted scenarios
  6. Implement the internals simply
  7. Decide: keep it, erase it, or iterate

- How to adapt this workflow to different languages and environments: Python, TypeScript, Go – the discipline is language-agnostic
- IDE and tooling habits that support FDD: what to configure, what to ignore, what to stop doing
- Team adoption: how to introduce FDD practices incrementally without disrupting existing workflows
- The "starting somewhere" principle: a workflow that produces better designs 80% of the time, starting Monday, is more valuable than a perfect workflow that requires months to adopt
- What FDD looks like at scale: applying the same principles to system design, not just component design
- Next steps: what to read, what to build, and how to deepen the practice over time

Example FDD Workflow - The Five Whys Example

We start with a black box: one input, one output, and a second output for error messages.

The input port is named with the empty string `""`. The output port is also named `""`. The error port is named `"✗"`.

For now we ignore the error output and focus on the happy path – sending a question in and getting a sufficiently detailed answer out.

At this stage, both the problem and our solution are vague. This is a situation every programmer and consultant knows well: the customer senses a problem they want solved, but cannot yet express it in enough detail to write code. Daniel Pink, in his masterclass, recommends the *five whys* technique to drill past this vagueness. Each time the customer gives an answer, you rephrase it as a new "why" question and repeat – five times. Each question is grounded in the previous answer rather than rehashed from scratch. The result tends to uncover enough concrete detail to actually write code.

We adopt this idea and build a tool that runs the five-whys process automatically using an LLM. The tool is developed using the FDD (Feature-Driven Development) approach, demonstrating the top-down / bottom-up spiral that refines a design from sketch to running code.

At the very top level we have one black box with one input, one answer output, and one error output. It behaves like a function, but not quite: there is a measurable delay between input and output, and the two outputs are mutually exclusive – if the answer fires, the error port produces nothing at all, and vice versa.

We can sketch this as:

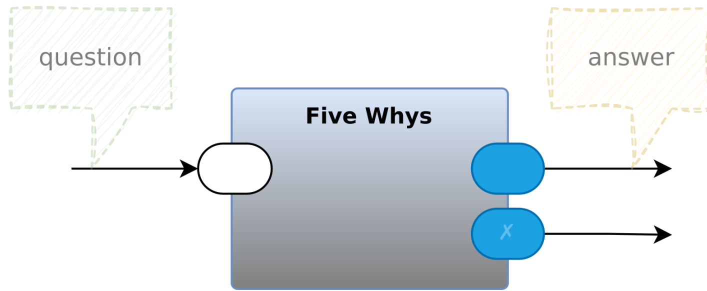
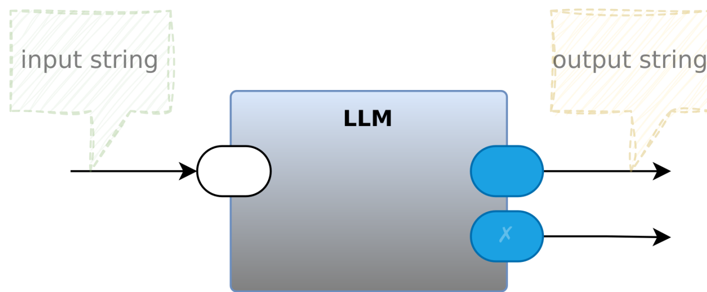
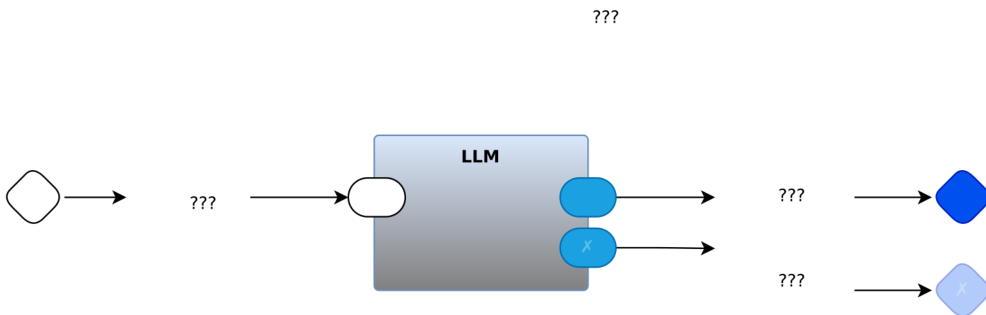


Figure 1. top level

That is deliberately vague. We know the box contains an LLM somewhere inside, so our next step is to pin down what that LLM part looks like and make it concrete enough to add to our parts palette. Roughly:



That is not enough to build the rest of "Five Whys", but it is a start. We chip away at both ends simultaneously:



Getting the LLM Running

How do we fit a whole LLM into a single black box? If we can invoke it from the command line, we can wrap the command line in a "shell-out" part.

Rather than building an LLM from scratch, we borrow an existing one. We use agency, forked from neurocult/agency. It is written in Go (not Python), which demonstrates that we can freely mix languages at this stage without worrying about uniformity or efficiency. It runs from the command line and calls the OpenAI API, for which you need a valid API key.

First, we build the *agency* source and confirm it runs from the command line. We install dependencies with `./@install` (done once), then run `./@make`, which sets shell variables and calls `./@makec` and `./@testc`.

`@testc` invokes the LLM directly:

```
1  echo "is concurrency considered difficult?" | ./agency/main -model gpt-4o-mini
   ↪  -maxTokens 1000 -temp=1 -prompt "concise responses"
```

For a shell-out part, input must arrive on `stdin` and output must go to `stdout` – the standard Unix pipeline convention. *Agency* handles this naturally: the `echo` above pipes the question into `stdin`, and the rest of the command line configures the model.

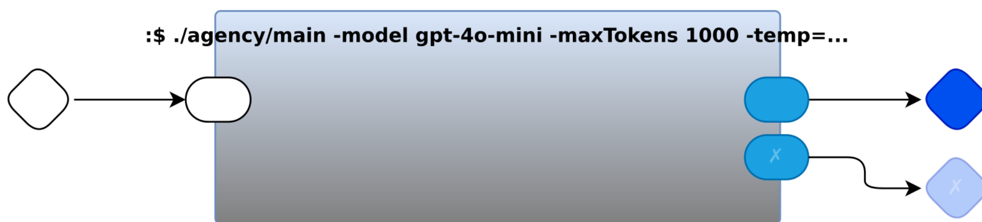
The successful build and test:

```
-zsh                                                                    ⌥⌘1
$ ./@install

up to date, audited 27 packages in 684ms

3 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
$ ./@make
go build examples/cli/main.go
Yes, concurrency can be difficult due to issues like race conditions, deadlocks, and the
complexity of coordinating multiple threads or processes. Proper synchronization and unde
rstanding of parallelism are essential.
$ 
```

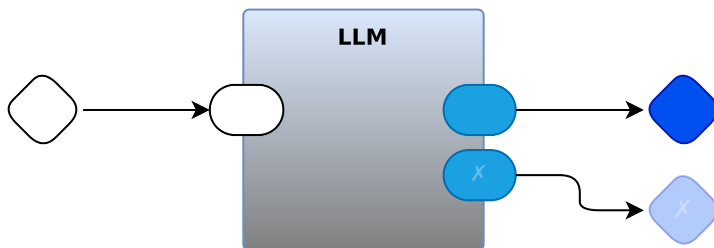
Wrapping the LLM as a Part

To turn the command line into a Leaf part, we prefix it with `:$`:

For readability we wrap that inside a Container part called "LLM" – a new tab in the drawio file:

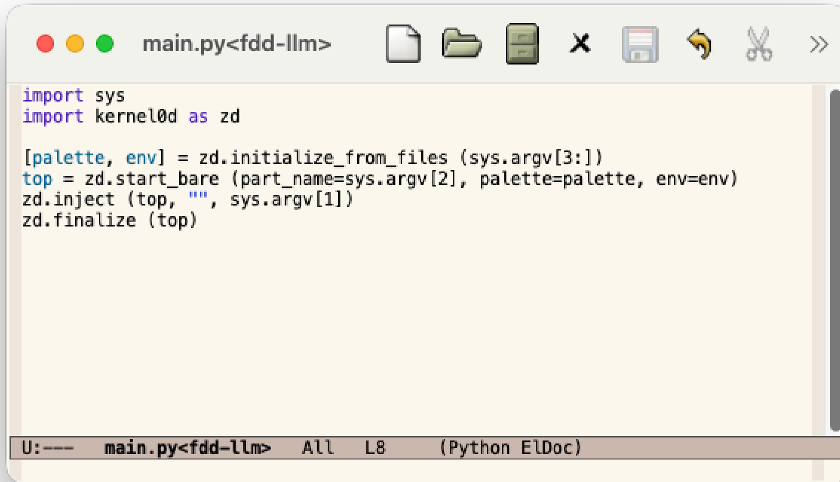


The simplest test jig is another Container part that feeds its input port straight into the LLM:

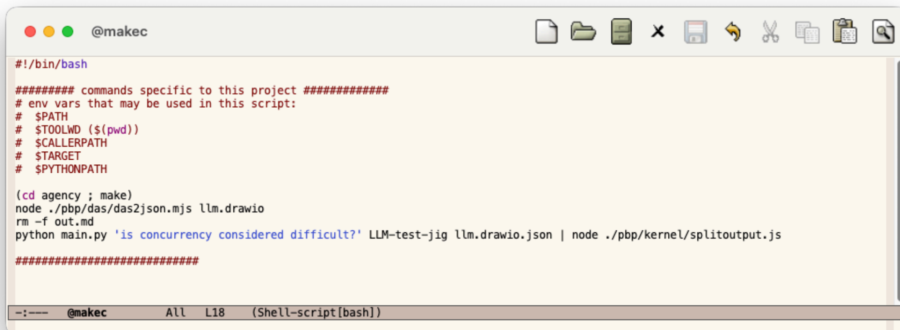


With this jig we write a small Python coupler and pass one string argument on the command line, watching the output on the terminal.

The coupler code:



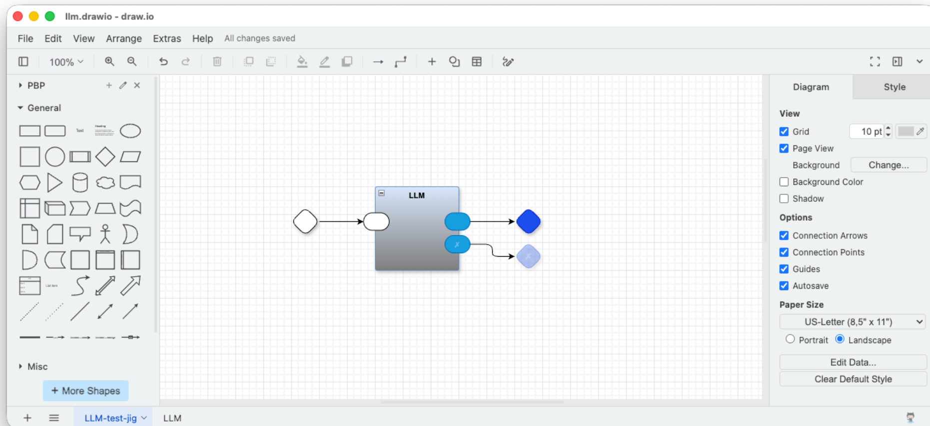
`@makec` extended to run it:

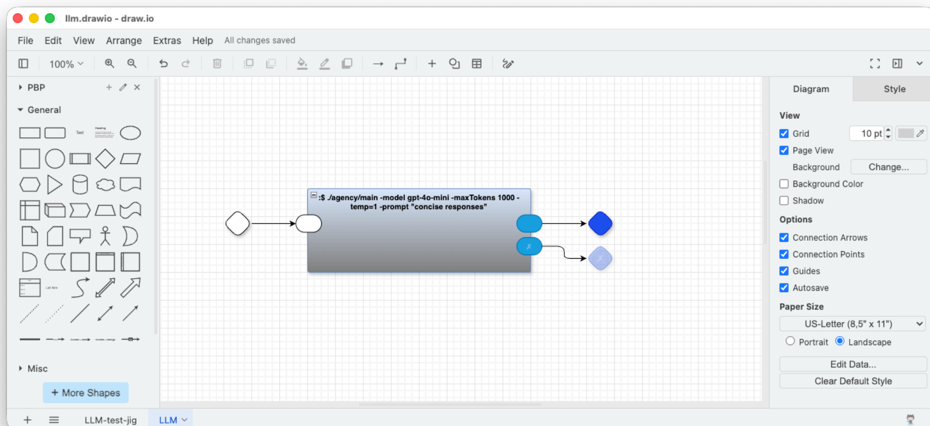


The `das2json` line converts `llm.drawio` to `llm.drawio.json`, stripping graphics noise. The `python main.py` line runs the PBP tool with `LLM-test-jig` as the top-level container and `'is concurrency considered difficult?'` as the input string.

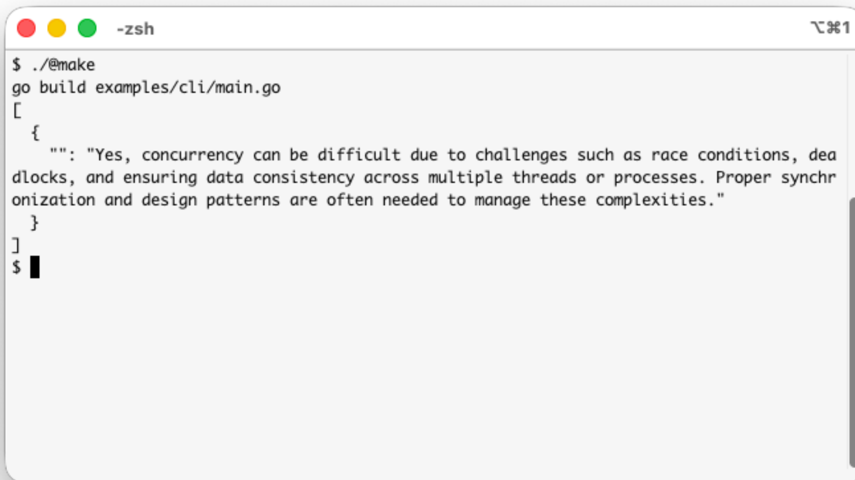
We also empty out `@testc`, since the LLM part now calls the OpenAI API rather than running agency directly from a command line.

We create the test jig in drawio with two tabs – `LLM-test-jig` and `LLM`. Shell-out parts are recognised by the PBP system and need no additional tab or Python file.





Running the test jig:



The shell-out part calls *agency*, correctly nested inside `LLM`, which calls the OpenAI API and returns the answer. The output from a PBP top-level part is a JSON array of tagged strings – key/value pairs where the key is the output port name (here `""`) and the value is the LLM response.

What I Learned From This Test Jig

The answer contains several distinct issues rather than one. That means I will need to break the answer into multiple questions, or build a single composite question.

I also notice that each run takes a second or two and costs a small amount of money. If I am conscious of cost I may hesitate to run tests freely during iteration – which defeats the purpose of quick iteration. This immediately suggests I should stub out the API call so I can run as many tests as I like for free.

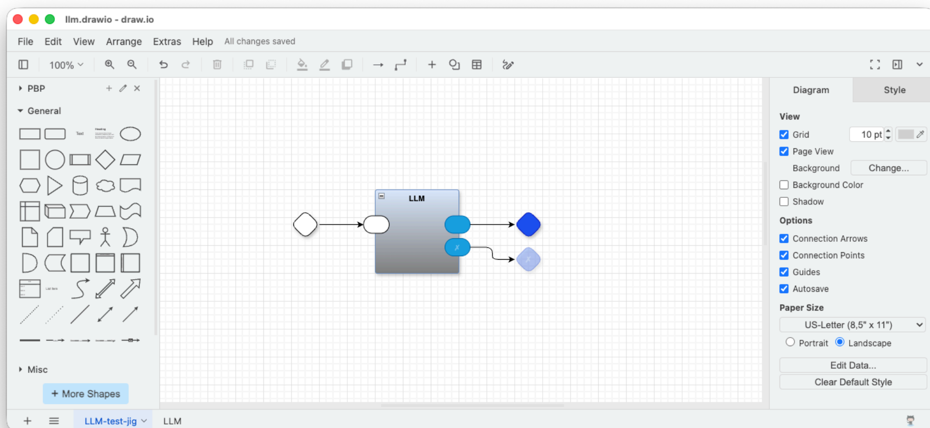
Neither observation is surprising in hindsight, but building the test jig surfaced them early, when they are cheapest to address.

I now have one reusable part on my palette – the LLM part. I do not expect to revisit it. That slice of attention is freed up for the rest of the design.

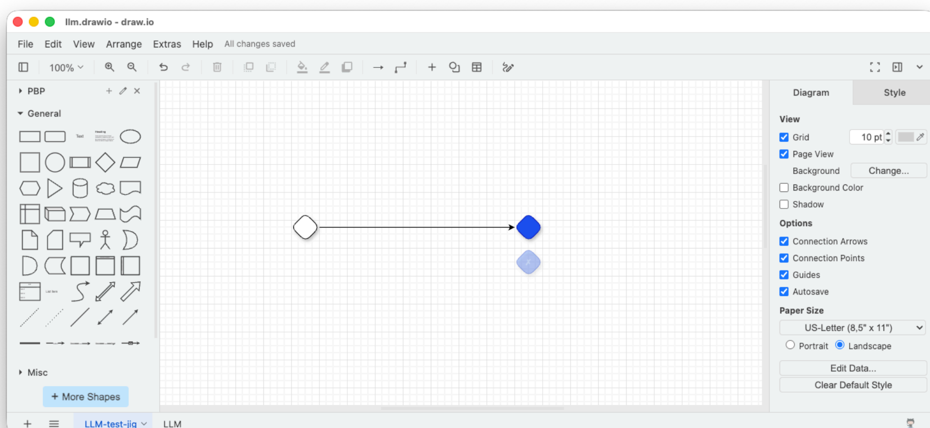
Build a Stub Test Jig (No API Calls)

The fix is trivial: replace the LLM part in the test jig with a “through” connection that passes input directly to output.

From:



To:



The hanging error port is harmless – if nothing is ever sent to it, it never fires.

Any string sent into `LLM-test-jig` now passes straight through without touching the API:

Input: `"is concurrency considered difficult?"`. Output: exactly the same string. As expected.

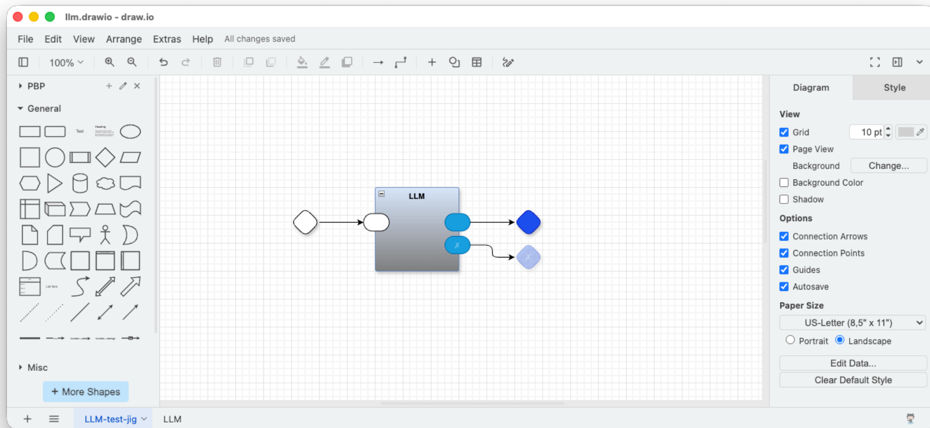
Count To Five

Next we need a Leaf part that counts incoming answers. The first four go to output port `"<5"` and the fifth goes to output port `"5th"`. We call it `counter` and write it in Python – a diagram would be overkill for something this small.

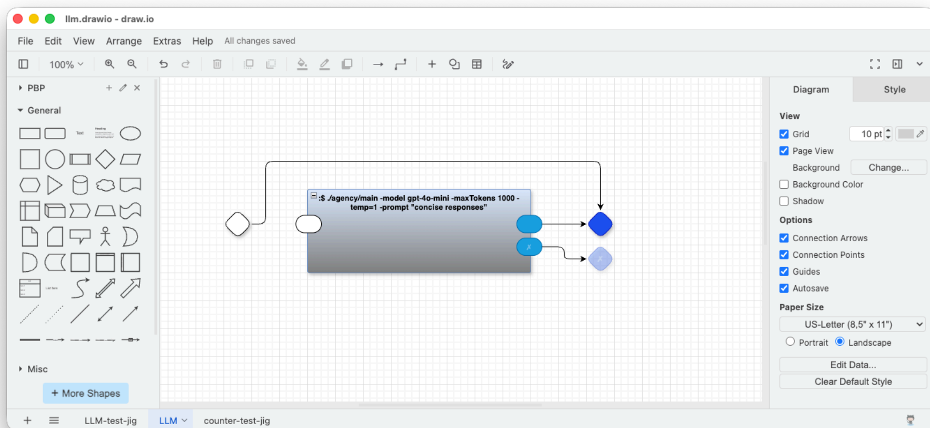
Rethink and Retrench

While devising a test jig for the counter, I realise the "through" stub would sit better inside the LLM part itself, not in the test jig. Moving it there lets the original test jig remain unchanged and makes the stub easier to toggle.

I reset the first test jig:



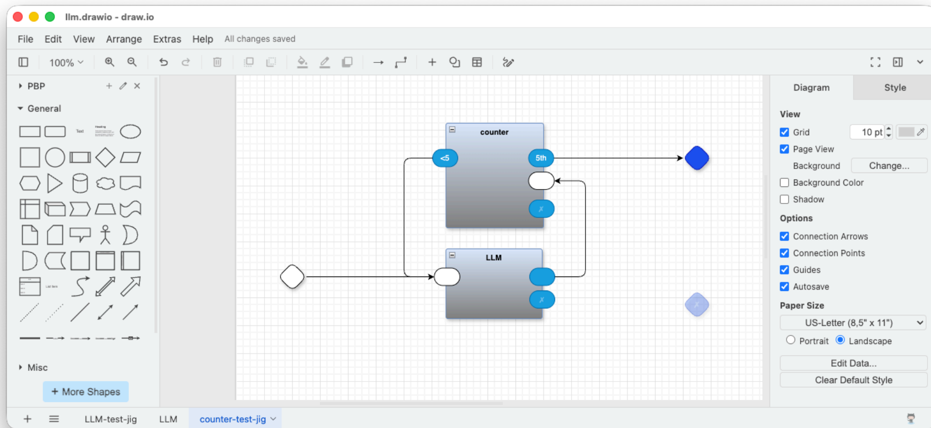
And change the LLM part internally to be a through-connection:



Running `./@make` confirms the output is still identical to the input. This is a design iteration – trivial to make, and it sets up future tests more cleanly.

Continue Count To Five

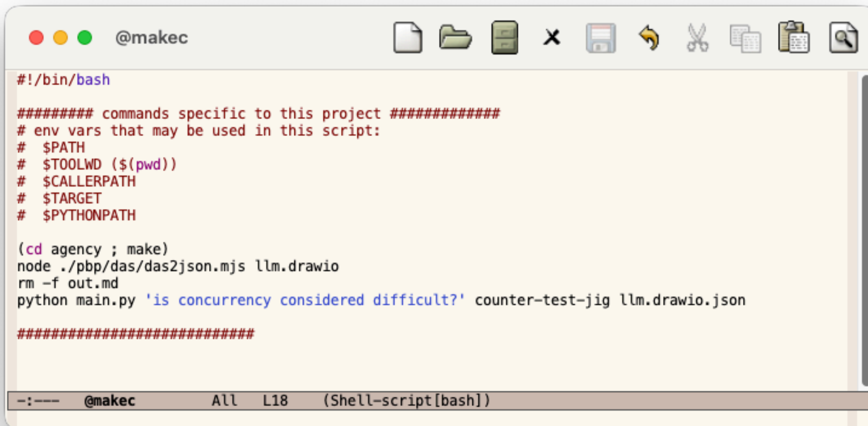
I create a test jig for the counter:



And write `counter.py`:

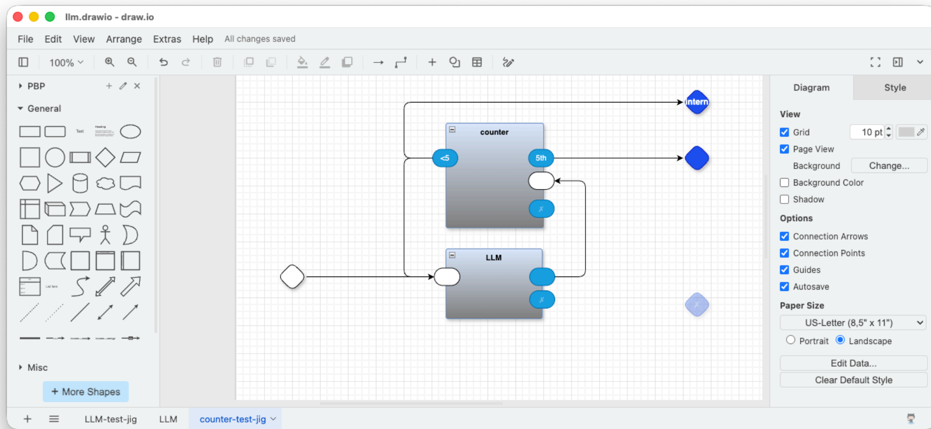
```
class Counter:
    def __init__ (self):
        self.max = 5
        self.count = 1

    def inc (self):
        self.count += 1

def handler (eh, mev):
    self = eh.instance_data
    self.inc ()
    if self.count < self.max:
        zd.send (eh, "<5", mev.datum.v, mev)
    else:
        zd.send (eh, "5th", mev.datum.v, mev)

def instantiate (reg, owner, name, arg, template_data):
    name_with_id = zd.gensymbol ("counter")
    self = Counter ()
    return zd.make_leaf (name_with_id, owner, self, arg, handler, None)

def install (reg):
    zd.register_component (reg, zd.mkTemplate ("counter", None, instantiate))

-:---  counter.py      Bot    L29     (Python ElDoc)
Wrote /Users/paultarvydas/projects/fdd-llm/counter.py
```

I update `@makec` to point at the new top-level diagram:



Injecting one question, I expect to see it echoed verbatim in the output:

That works. For more confidence, I want to watch the feedback loop. One option is a dedicated internal output port:



Which gives:

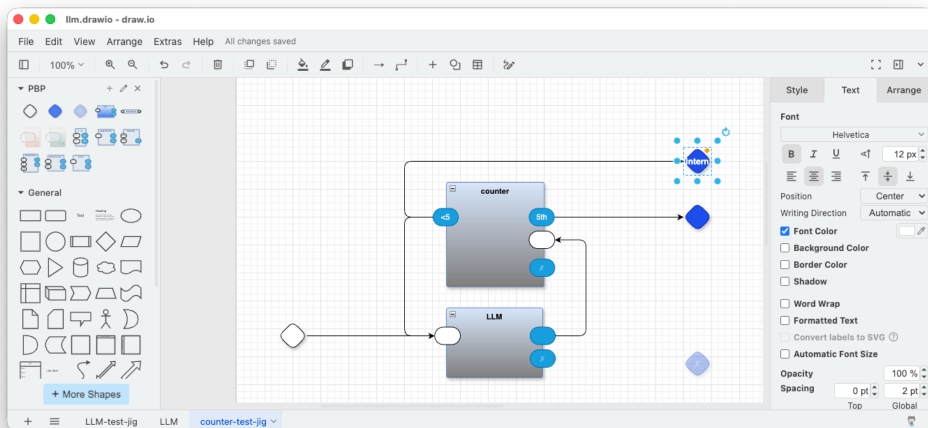
```
-zsh                                                        ⌥⌘1
$ ./@make
go build examples/cli/main.go
[
  {
    "         <div>intern</div>": "is concurrency considered difficult?"
  },
  {
    "         <div>intern</div>": "is concurrency considered difficult?"
  },
  {
    "         <div>intern</div>": "is concurrency considered difficult?"
  },
  {
    "         <div><br/></div>": "is concurrency considered difficult?"
  }
]
$ █
```

New Insights, Pass 2

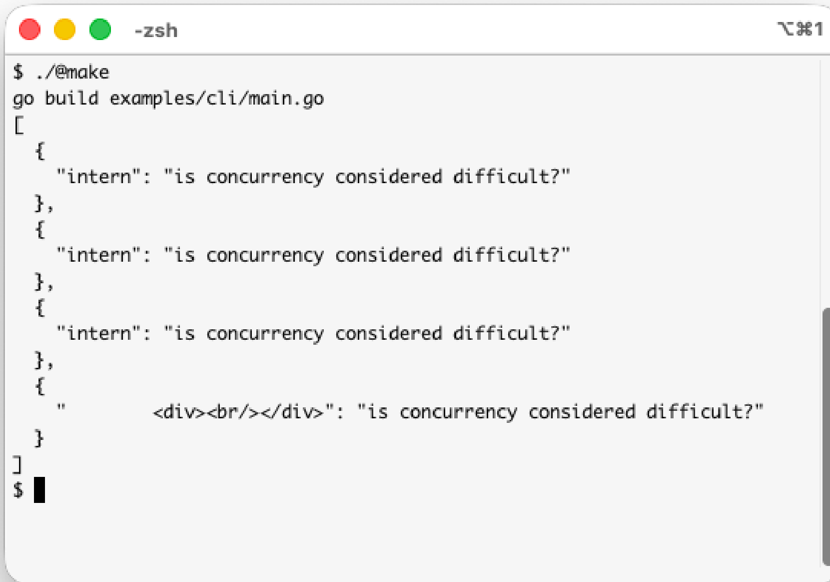
Two things jump out immediately:

1. The counter exhibits an off-by-one error – it counts to 3 instead of 4.
2. I forgot to turn off “Formatted Text” in the drawio editor, so the output contains raw `<div>` markup.

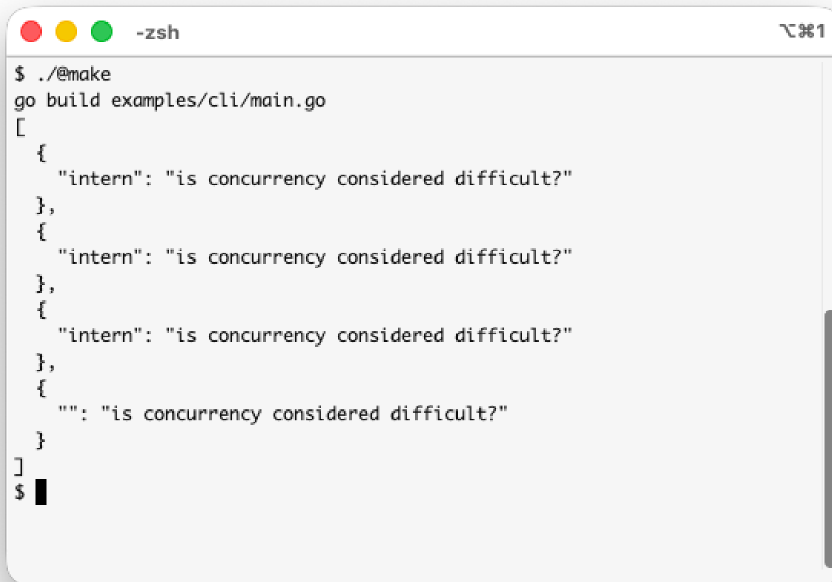
Turn Off Formatted Text



I click the “intern” output port, open the Text tab in the properties panel, and uncheck “Formatted Text”. After that:

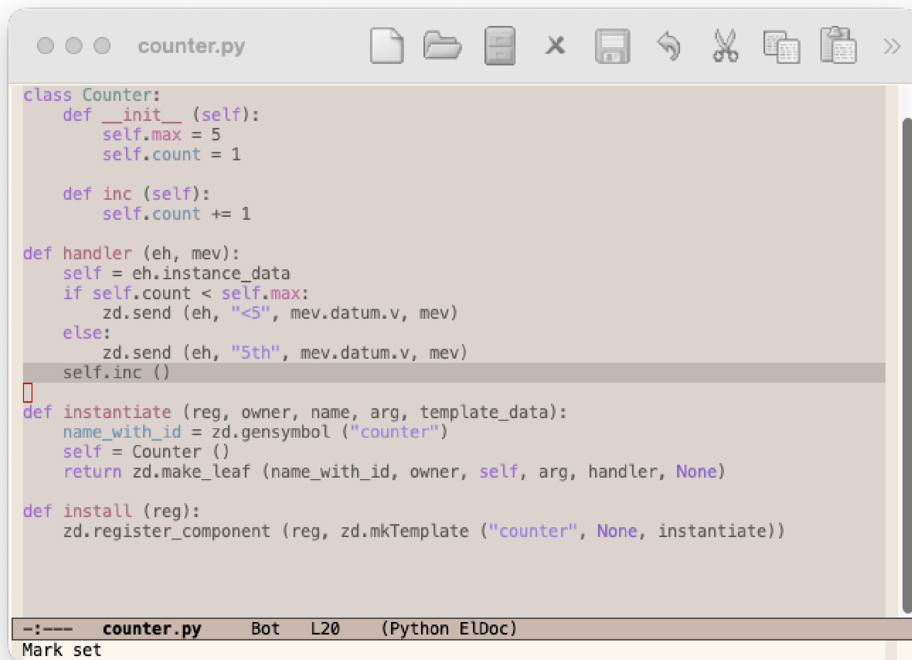


That fixes the first three outputs, but the final `""` output still has the markup. Selecting all objects and turning off Formatted Text everywhere – and checking that all output port names are clean empty strings with no spaces – fixes it:



Fix the Off-By-One Error

Moving the `inc()` call to the right place in `counter.py`:

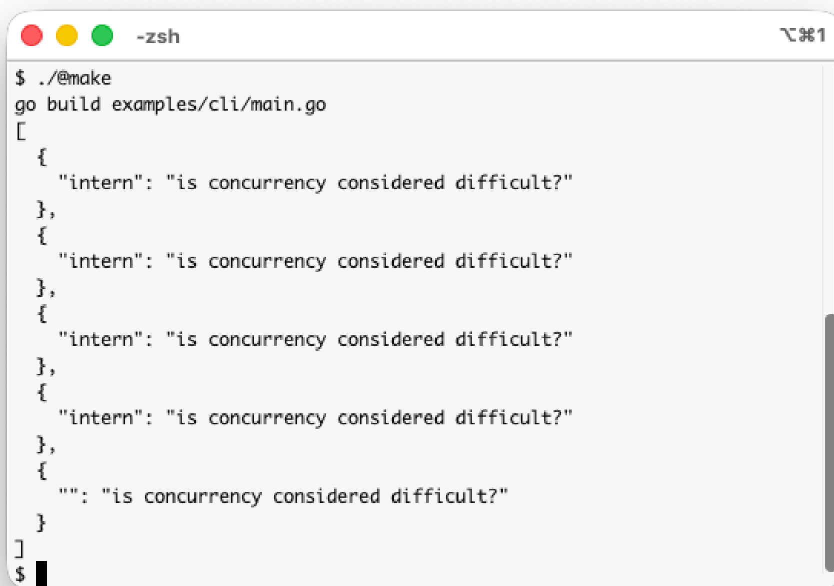

counter.py
class Counter:
    def __init__ (self):
        self.max = 5
        self.count = 1

    def inc (self):
        self.count += 1

def handler (eh, mev):
    self = eh.instance_data
    if self.count < self.max:
        zd.send (eh, "<5", mev.datum.v, mev)
    else:
        zd.send (eh, "5th", mev.datum.v, mev)
    self.inc ()

def instantiate (reg, owner, name, arg, template_data):
    name_with_id = zd.gensymbol ("counter")
    self = Counter ()
    return zd.make_leaf (name_with_id, owner, self, arg, handler, None)

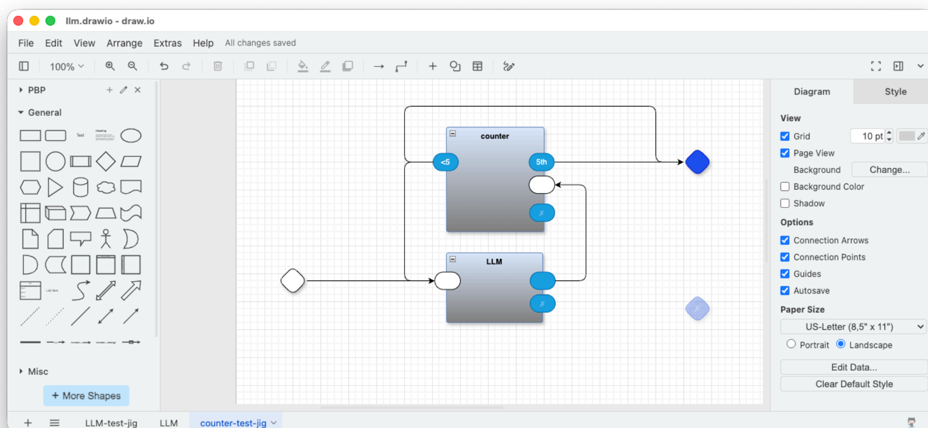
def install (reg):
    zd.register_component (reg, zd.mkTemplate ("counter", None, instantiate))
-:--- counter.py Bot L20 (Python ElDoc)
Mark set



Different Ways to Probe

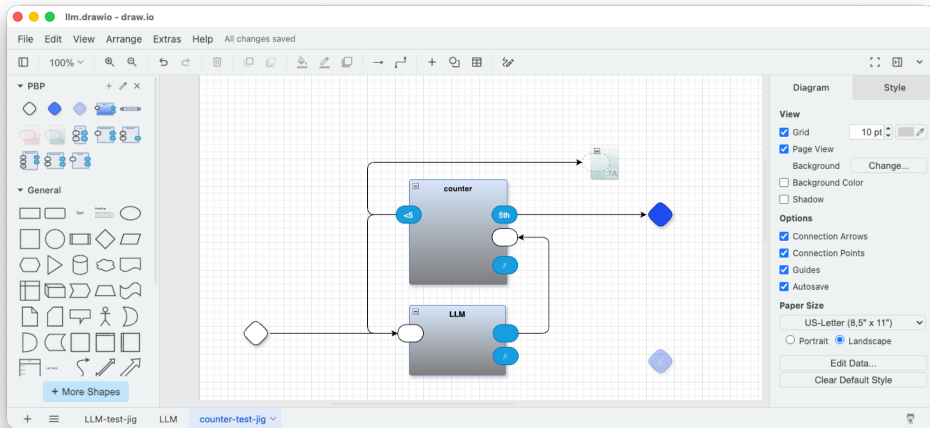
Traditional debugging relies on `printf` statements, sequential debuggers, and watchpoints. PBP's fan-out gives us additional options that are very easy to set up and remove.

Route to the default output port – split `"<5"` and send both copies to `""`:




```
-zsh                                                        ⌥⌘1
$ ./@make
go build examples/cli/main.go
[
  {
    "": "is concurrency considered difficult?"
  },
  {
    "": "is concurrency considered difficult?"
  },
  {
    "": "is concurrency considered difficult?"
  },
  {
    "": "is concurrency considered difficult?"
  },
  {
    "": "is concurrency considered difficult?"
  }
]
$ █
```

Use probe parts – tap off an output without redirecting it, much like an oscilloscope probe on a circuit:




```
-zsh
$ ./@make
go build examples/cli/main.go
"Info" : "  @4  probe counter-test-jig▸counter-test-jig▸:?A2: is concurrency considered difficult?"
"Info" : "  @7  probe counter-test-jig▸counter-test-jig▸:?A2: is concurrency considered difficult?"
"Info" : "  @10  probe counter-test-jig▸counter-test-jig▸:?A2: is concurrency considered difficult?"
"Info" : "  @13  probe counter-test-jig▸counter-test-jig▸:?A2: is concurrency considered difficult?"
[
  {
    "": "is concurrency considered difficult?"
  }
]
$
```

Four probe outputs appear, followed by the final output. All strings are identical (the LLM is still stubbed). The probe format is:

```
1  "Info" : "  @<N>  probe <fully-qualified-name>?<probe-name><id>: <payload>"
```

- `@<N>` – internal tick count; useful for relative ordering (`@4` precedes `@7`)
- `<name>` – fully-qualified part name

- `?<probe-name>` – the probe's label in the diagram (default `?A`)
- `<id>` – unique numeric kernel ID
- `<payload>` – the data carried by the message

For example: `"Info" : " @4 probe counter-test-jig▹counter-test-jig▹:?A`$_2$`: is concurrency considered difficult?"` is a debug message at tick 4 with payload `"is concurrency considered difficult?"`.

Probes are internally given higher priority than other messages, so they always print before the parts further along the same fan-out chain – a small kernel detail that keeps probe output in a sensible order.

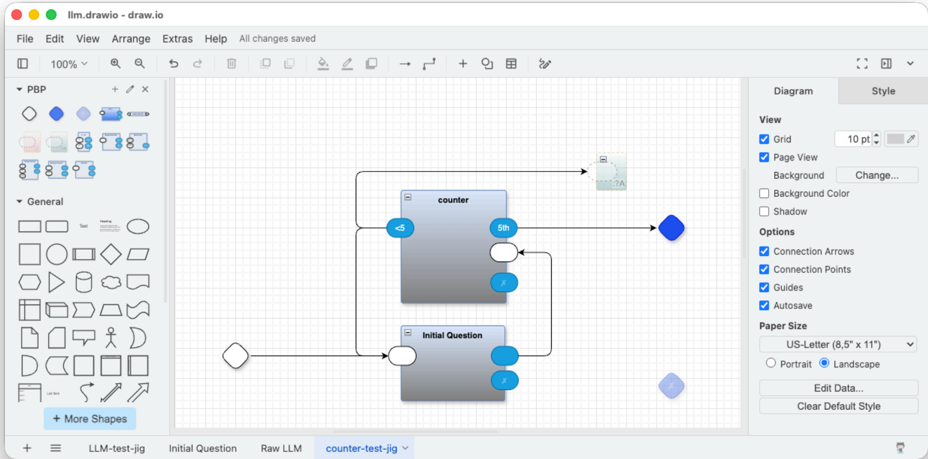
In electronics, probing draws a tiny current from the circuit (impedance matters). In software, probing copies a message (fan-out). In the 21st century that copy is essentially free, so we can probe anywhere we like.

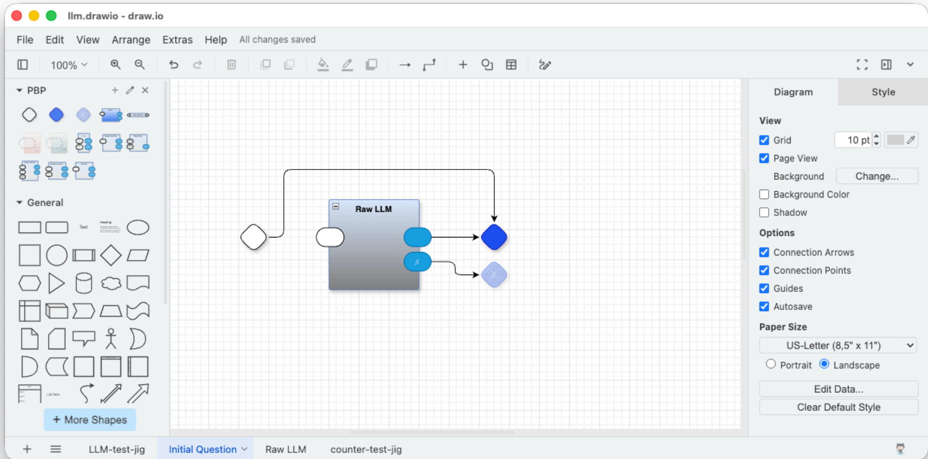
Internally Rephrasing Answers as Questions

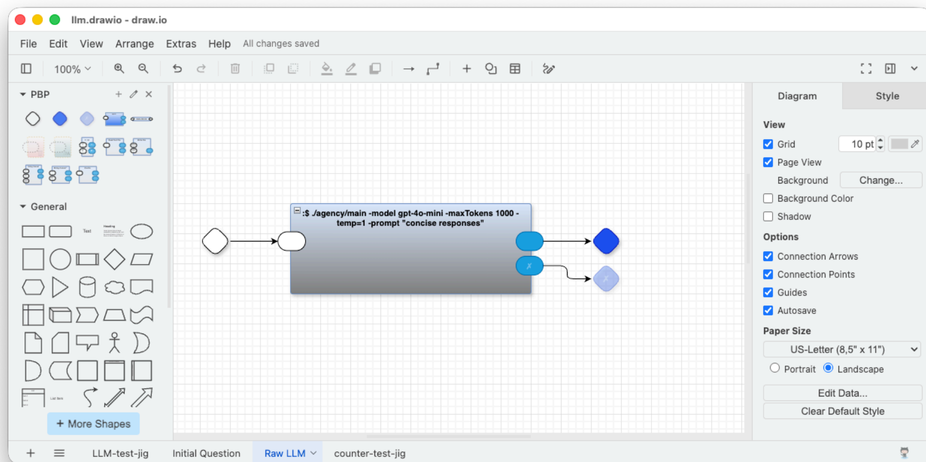
The next step is to rephrase each LLM answer as a new question and feed it back into the loop. The easiest way is a second instance of the LLM part, with a slightly different prompt. Before wiring that up, I add one more Container wrapper around the raw shell-out so I can configure each LLM call independently.

One More Wrapper

I rename parts and restructure the counter test jig:


llm.drawio - draw.io
File Edit View Arrange Extras Help All changes saved
100%
PBP
General
+ More Shapes
counter
<5
5th
Initial Question
Diagram
Style
View
Grid
10 pt
Page View
Background
Change...
Background Color
Shadow
Options
Connection Arrows
Connection Points
Guides
Autosave
Paper Size
US-Letter (8,5" x 11")
Portrait
Landscape
Edit Data...
Clear Default Style
LLM-test-jig
Initial Question
Raw LLM
counter-test-jig


llm.drawio - draw.io
File Edit View Arrange Extras Help All changes saved
100%
PBP
General
+ More Shapes
Raw LLM
Diagram
Style
View
Grid
10 pt
Page View
Background
Change...
Background Color
Shadow
Options
Connection Arrows
Connection Points
Guides
Autosave
Paper Size
US-Letter (8,5" x 11")
Portrait
Landscape
Edit Data...
Clear Default Style
LLM-test-jig
Initial Question
Raw LLM
counter-test-jig



The original "LLM" part becomes "Raw LLM". A new "Initial Question" part wraps it but remains stubbed through for now. The top level of `counter-test-jig` uses "Initial Question" instead of "LLM".

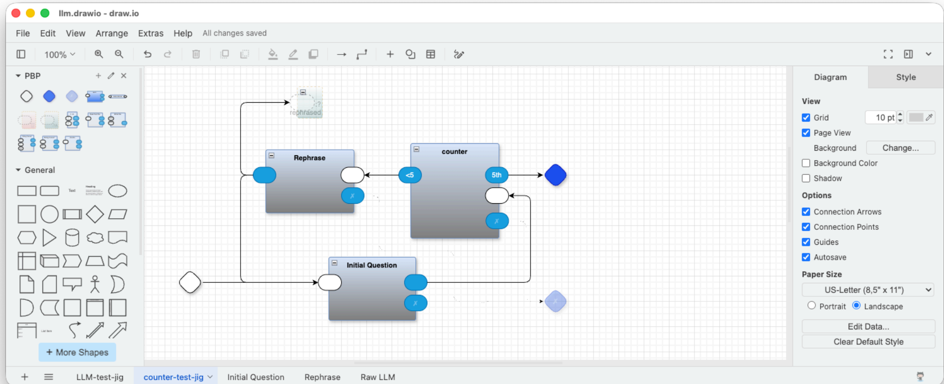
New Part For Rephrasing Answers

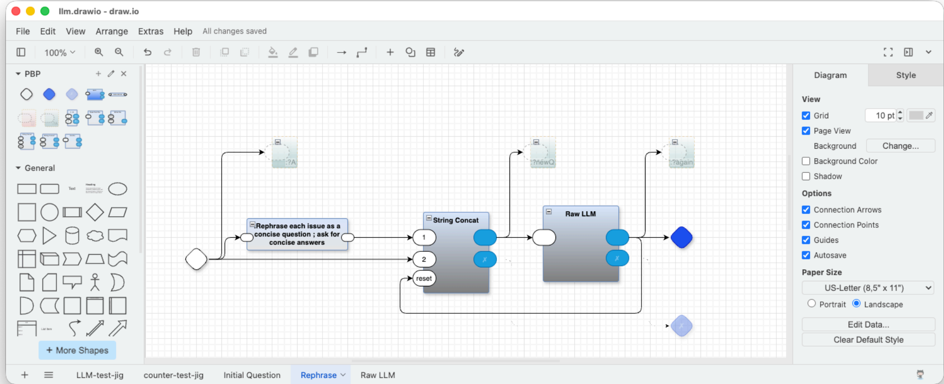
I create a "Rephrase" part and wire it into a feedback loop. When Rephrase receives input, it prepends the string `"Rephrase each issue as a concise question ; ask for concise answers"` to it, sends the result to Raw LLM, and forwards the LLM's output to its own output.

The output of Rephrase feeds back into "Initial Question" inside `counter-test-jig`.

A note on feedback loops: this is not recursion. There is a real time delay, and the feedback message goes to the *end* of "Initial Question"'s input queue rather than to the front as recursion would. Fan-out and queuing, not recursion.

I insert several probes to observe what is happening:





Running it:

Something is wrong. `das2json` warns of an unconnected port. The intent was to prepend the canned prompt string to the LLM answer using a "String Concat" part, then send the concatenated string to Raw LLM, and also send the Raw LLM output back to "reset" String Concat (to prepare it for the next iteration). The fan-out here sends one output to two destinations simultaneously.

The problem is a drawio quirk: drawio is a *drawing* editor, not a *programming* editor. It let me create a port that visually appeared attached to the String Concat part but was not actually attached in the graph sense. The fix is to drag the port inward until drawio shows a purple highlight on the String Concat shape, confirming containment. This is a drawing-editor gesture, not a PBP semantic – we are using drawio as a best-available tool and must be careful to satisfy its graphical rules so that PBP semantics can be correctly inferred.

After fixing the attachment, the feedback loop comes to life:

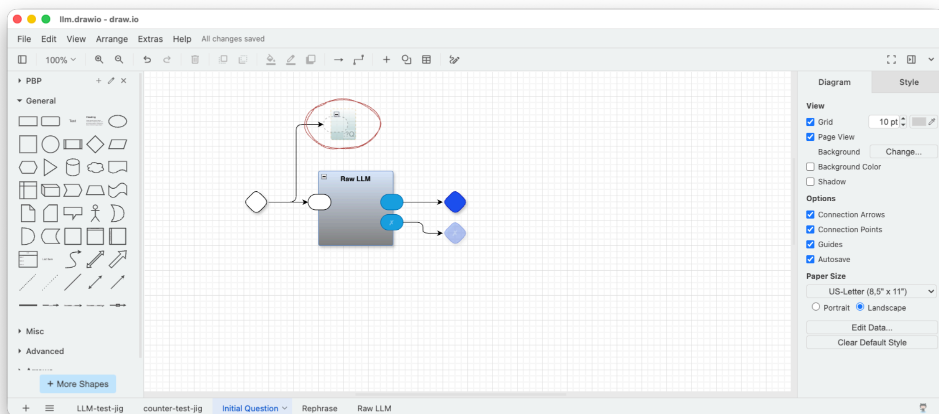
```
-zsh                                                                                               ⌥⌘1
5. How do context switching and locking impact performance in concurrent processes?"
"Info" : "  @54  probe counter-test-jig▸Initial Question▸counter-test-jig▸Initial Question▸:?Q₁: 1. What are race conditions, and how do they affect
 shared resources?

2. What are deadlocks, and how do they occur between processes?

3. How does concurrency add complexity to programming?

4. What are the main challenges of synchronization in concurrent systems?

5. How do context switching and locking impact performance in concurrent processes?"
[
  {
    "": "1. Race conditions occur when multiple processes or threads access shared resources simultaneously, leading to unpredictable outcomes. They
 can corrupt data and disrupt functionality.\n\n2. Deadlocks happen when two or more processes are each waiting for the other to release resources,
causing them to be stuck indefinitely. This typically occurs with circular resource dependencies.\n\n3. Concurrency adds complexity by introducing i
ssues like race conditions, deadlocks, and increased difficulty in managing shared state. It requires careful design and testing.\n\n4. Main challen
ges of synchronization include ensuring consistency, avoiding deadlocks, minimizing overhead, and managing resource contention efficiently.\n\n5. Co
ntext switching can degrade performance due to the overhead of saving and restoring process states. Locking can introduce delays and contention, slo
wing down execution if not managed properly."
  }
]
$
```

(Only the last screenful of output is shown; the full run follows below.)

* * *

Loop 1

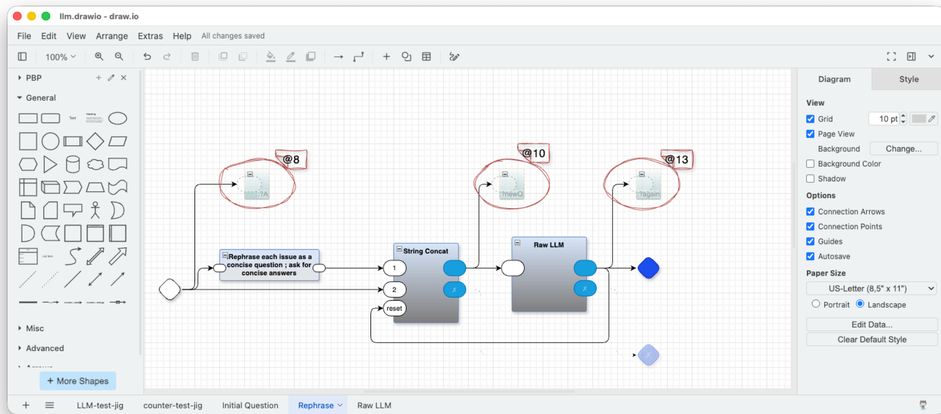
Initial question from the `@makec` script:



```
$ ./@make
go build examples/cli/main.go
"Info" : "  @2  probe counter-test-jig▸Initial
↪  Question▸counter-test-jig▸Initial Question▸:?Q1: is concurrency considered
↪  difficult?"
```

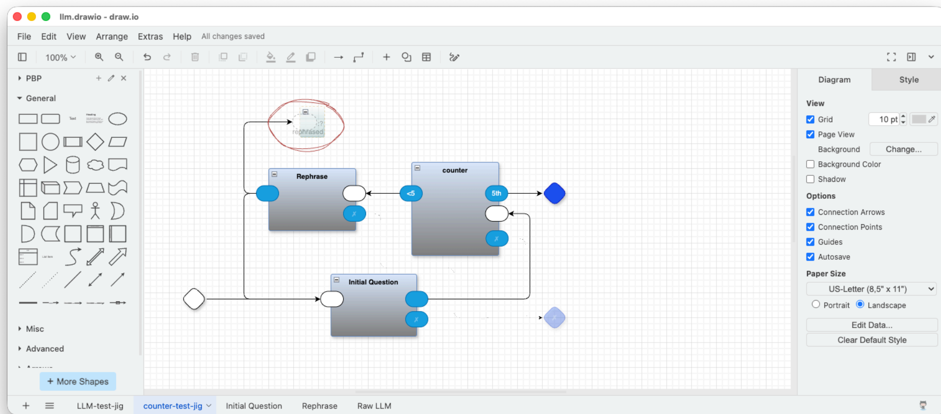
* * *

Probes “A”, “newQ”, and “again” fire:




```
"Info" : "  @8  probe counter-test-jig▸Rephrase▸counter-test-jig▸Rephrase▸:?A6:
↪  Yes, concurrency is often considered difficult due to challenges like
↪  synchronization, race conditions, deadlocks, and designing efficient
↪  algorithms for concurrent execution."
"Info" : "  @10  probe
↪  counter-test-jig▸Rephrase▸counter-test-jig▸Rephrase▸:?newQ7: Rephrase each
↪  issue as a concise question ; ask for concise answersYes, concurrency is
↪  often considered difficult due to challenges like synchronization, race
↪  conditions, deadlocks, and designing efficient algorithms for concurrent
↪  execution."
"Info" : "  @13  probe
↪  counter-test-jig▸Rephrase▸counter-test-jig▸Rephrase▸:?again8: What makes
↪  concurrency difficult?"
```

* * *




```
"Info" : "  @14  probe counter-test-jig▹counter-test-jig▹:?rephrased₉: What makes concurrency difficult?"
```

*　*　*

The initial response listed four issues, but the rephrasing collapsed them into a single question. A more precise prompt would produce one question per issue. We note this and continue.

*　*　*

Loop 2

The rephrased question feeds back into “Initial Question”:

```
"Info" : "  @15  probe counter-test-jig▹Initial Question▹counter-test-jig▹Initial Question▹:?Q₁: What makes concurrency difficult?"
```

*　*　*

Raw text entering “Rephrase”:

```
"Info" : "  @21  probe
    counter-test-jig▹Rephrase▹counter-test-jig▹Rephrase▹:?A₆: Concurrency is
    difficult due to several factors:

1. **Race Conditions**: Multiple threads accessing shared resources can lead to
    inconsistent or unexpected results.
2. **Deadlocks**: Processes may end up waiting indefinitely for resources held
    by each other.
3. **Complexity**: Tracking and managing multiple threads increases program
    complexity, making debugging more challenging.
4. **Synchronization**: Properly coordinating access to shared resources can be
    intricate and error-prone.
5. **Performance Issues**: Overhead from context switching and locking can
    hamper performance if not managed well."
```

* * *

After prepending the canned prompt:

```
"Info" : "  @23  probe
    counter-test-jig▹Rephrase▹counter-test-jig▹Rephrase▹:?newQ₇: Rephrase each
    issue as a concise question ; ask for concise answersConcurrency is
    difficult due to several factors:
...
```

Note the missing space at the join point ("`answersConcurrency`"). The LLM works around it, but it is worth fixing. For now we leave it.

* * *

The newly generated set of questions:

```
"Info" : "  @26  probe
↪  counter-test-jig▹Rephrase▹counter-test-jig▹Rephrase▹:?again₈: 1. How do
↪  race conditions affect thread access to shared resources?
11. What causes deadlocks in concurrent processes?
12. How does concurrency increase program complexity?
13. Why is synchronization of access to shared resources challenging?
14. How can performance issues arise from context switching and locking in
↪  concurrency?"
```

* * *

```
"Info" : "  @27  probe counter-test-jig▹counter-test-jig▹:?rephrased₉: 1. How
↪  do race conditions affect thread access to shared resources?
11. What causes deadlocks in concurrent processes?
12. How does concurrency increase program complexity?
13. Why is synchronization of access to shared resources challenging?
14. How can performance issues arise from context switching and locking in
↪  concurrency?"
```

* * *

Loop 3

Questions feed back into “Initial Question”:

```
"Info" : "  @28  probe counter-test-jig▹Initial
↪  Question▹counter-test-jig▹Initial Question▹:?Q₁: 1. How do race conditions
↪  affect thread access to shared resources?
15. What causes deadlocks in concurrent processes?
16. How does concurrency increase program complexity?
17. Why is synchronization of access to shared resources challenging?
18. How can performance issues arise from context switching and locking in
↪  concurrency?"
```

* * *

“Initial Question” answers and sends the result to “counter”, which forwards to “Rephrase” (third “why” loop):

```
1  "Info" : "  @34  probe
   ↪ counter-test-jig▹Rephrase▹counter-test-jig▹Rephrase▹:?A₆: 1. Race
   ↪ conditions occur when multiple threads access shared resources
   ↪ simultaneously and at least one thread modifies the resource, leading to
   ↪ inconsistent or unpredictable results.
2
3  19. Deadlocks are caused by two or more processes holding resources while
   ↪ waiting for each other to release what they need, resulting in a circular
   ↪ wait situation.
4
5  20. Concurrency increases program complexity by introducing challenges like
   ↪ synchronization, shared state management, and the potential for bugs (e.g.,
   ↪ race conditions and deadlocks).
6
7  21. Synchronization is challenging because it requires careful coordination of
   ↪ access to resources, often involving complex logic to avoid deadlocks and
   ↪ ensure data integrity, while minimizing performance impacts.
8
9  22. Performance issues can arise from context switching (overhead of switching
   ↪ between threads) and locking (potentially causing threads to wait,
   ↪ increasing latency and reducing throughput)."
```

*   *   *

```
1  "Info" : "  @39  probe
   ↪ counter-test-jig▹Rephrase▹counter-test-jig▹Rephrase▹:?again₈: 1. What are
   ↪ race conditions and how do they affect shared resources?
2
3  27. How do deadlocks occur in processes?
4
5  28. In what ways does concurrency increase program complexity?
6
7  29. Why is synchronization considered challenging in programming?
8
9  30. What performance issues are associated with context switching and locking?"
```

*   *   *

```
1 "Info" : "  @40  probe counter-test-jig▹counter-test-jig▹:?rephrased₉: 1. What
  ↪  are race conditions and how do they affect shared resources?
2
3 6. How do deadlocks occur in processes?
4
5 7. In what ways does concurrency increase program complexity?
6
7 8. Why is synchronization considered challenging in programming?
8
9 9. What performance issues are associated with context switching and locking?"
```

* * *

Loop 4

```
1 "Info" : "  @41  probe counter-test-jig▹Initial
  ↪  Question▹counter-test-jig▹Initial Question▹:?Q₁: 1. What are race
  ↪  conditions and how do they affect shared resources?
2
3 10. How do deadlocks occur in processes?
4
5 11. In what ways does concurrency increase program complexity?
6
7 12. Why is synchronization considered challenging in programming?
8
9 13. What performance issues are associated with context switching and locking?"
```

* * *

```
1  "Info" : "  @52  probe
   ↪  counter-test-jig▹Rephrase▹counter-test-jig▹Rephrase▹:?again8: 1. What are
   ↪  race conditions, and how do they affect shared resources?
2
3  6. What are deadlocks, and how do they occur between processes?
4
5  7. How does concurrency add complexity to programming?
6
7  8. What are the main challenges of synchronization in concurrent systems?
8
9  9. How do context switching and locking impact performance in concurrent
   ↪  processes?"
```

* * *

```
1  "Info" : "  @53  probe counter-test-jig▹counter-test-jig▹:?rephrased9: 1. What
   ↪  are race conditions, and how do they affect shared resources?
2
3  6. What are deadlocks, and how do they occur between processes?
4
5  7. How does concurrency add complexity to programming?
6
7  8. What are the main challenges of synchronization in concurrent systems?
8
9  9. How do context switching and locking impact performance in concurrent
   ↪  processes?"
```

* * *

Final Output

The questions feed into “Initial Question” one last time:

```
"Info" : "  @54  probe counter-test-jig▹Initial Question▹counter-test-jig▹Initial Question▹:?Q₁: 1. What are race conditions, and how do they affect shared resources?

10. What are deadlocks, and how do they occur between processes?

11. How does concurrency add complexity to programming?

12. What are the main challenges of synchronization in concurrent systems?

13. How do context switching and locking impact performance in concurrent processes?"
```

* * *

This time the counter sends the answers to the final output instead of back through the loop:

```
[
  {
    "": "1. Race conditions occur when multiple processes or threads access shared resources simultaneously, leading to unpredictable outcomes. They can corrupt data and disrupt functionality.\n\n2. Deadlocks happen when two or more processes are each waiting for the other to release resources, causing them to be stuck indefinitely. This typically occurs with circular resource dependencies.\n\n3. Concurrency adds complexity by introducing issues like race conditions, deadlocks, and increased difficulty in managing shared state. It requires careful design and testing.\n\n4. Main challenges of synchronization include ensuring consistency, avoiding deadlocks, minimizing overhead, and managing resource contention efficiently.\n\n5. Context switching can degrade performance due to the overhead of saving and restoring process states. Locking can introduce delays and contention, slowing down execution if not managed properly."
  }
]
$
```

Make the Debug Output More Concise

The feedback loop is working, but the probe output is noisy enough that it is hard to evaluate the quality of the rephrasing. The probes for “Q”, “A”, “newQ”,

and "again" served their purpose – verifying that the loop runs correctly. Now I want to evaluate the *output*, so I remove those probes and keep only the "rephrased" probe. The machine regenerates all the intermediate results for free; I save the head-scratching.

Delete Redundant Probes

Probes "Q", "A", "newQ", and "again" are removed. Only "rephrased" remains. Final output continues to appear on the terminal as a matter of course.

Improve the Rephrase Prompt

The current prompt is: `":Rephrase each issue as a concise question ; ask for concise answers"`.

Rather than guessing at improvements, I run the test jig and see what the intermediate questions look like. That will give me concrete inspiration.

Top-Level Iteration and Analysis

```
1 $ ./@make
2 go build examples/cli/main.go
```

* * *

```
1 "Info" : "  @14  probe counter-test-jig▹counter-test-jig▹:?rephrased5: 1. How
  ↪  do race conditions complicate concurrency?
2 2. What challenges do deadlocks present in concurrent systems?
3 3. Why is proper synchronization essential in managing concurrency?"
```

* * *

```
1 "Info" : "  @27  probe counter-test-jig▹counter-test-jig▹:?rephrased5: 1. What
  ↪  are the consequences of race conditions in multi-threaded environments?
2
3 4. How do deadlocks affect system performance and progress?
4
5 5. Why is proper synchronization important for managing concurrency?"
```

* * *

```
1 "Info" : "  @40  probe counter-test-jig▹counter-test-jig▹:?rephrased5: 1. What
  ↪  negative effects can race conditions cause in multi-threaded environments?
2
3 6. How do deadlocks impact system performance and resource management?
4
5 7. Why is proper synchronization important for managing concurrency?"
```

* * *

```
1 "Info" : "  @53  probe counter-test-jig▹counter-test-jig▹:?rephrased5: 1. How
  ↪  do race conditions affect data integrity and behavior in multi-threaded
  ↪  environments?
2
3 8. What impact do deadlocks have on system performance and resource management?
4
5 9. Why is proper synchronization essential for managing concurrency?"
```

* * *

```
[
  {
    "": "1. Race conditions occur when multiple threads access shared data
↪   simultaneously, leading to unpredictable results and data corruption.
↪   Without proper control, the final outcome can depend on the timing of
↪   thread execution, compromising data integrity.\n\n2. Deadlocks can
↪   severely degrade system performance by causing processes to wait
↪   indefinitely for resources, leading to stalled applications. This can
↪   result in inefficient resource utilization and may require manual
↪   intervention to resolve, impacting overall resource management.\n\n3.
↪   Proper synchronization is essential for managing concurrency because it
↪   ensures that only one thread accesses critical sections of code or
↪   shared data at a time, preventing data corruption and ensuring
↪   consistent and predictable program behavior."
  }
]
$
```

Analysis

Four internal rephrasings at ticks 14, 27, 40, and 53, followed by the final output. That is correct – four “why” loops before the fifth answer exits.

Reformatted for readability:

1. Race conditions occur when multiple threads access shared data simultaneously, leading to unpredictable results and data corruption. Without proper control, the final outcome depends on the timing of thread execution, compromising data integrity.
2. Deadlocks can severely degrade system performance by causing processes to wait indefinitely for resources, leading to stalled applications. This results in inefficient resource utilisation and may require manual intervention, impacting overall resource management.
3. Proper synchronization is essential for managing concurrency because it ensures that only one thread accesses a critical section of code or shared data at a time, preventing data corruption and ensuring consistent, predictable behaviour.

* * *

Is the Prompt Good Enough?

Because we are using an LLM, exact repeatability is not guaranteed across runs. Looking at the `@14` rephrasing in this run, it looks reasonable. Rather than tuning the prompt speculatively, we will set this aside and revisit it only if multiple runs suggest a systematic weakness.

Does the Output Look Useful?

The initial question was `"is concurrency considered difficult?"`. The tool responded with three concrete points: race conditions, deadlocks, and synchronization. That is meaningfully less vague than the original. As a programmer or consultant trying to scope a product, reasoning about these three issues is far more productive than reasoning about "difficulty" in the abstract.

A Biased Note

From experience building compilers and operating systems: all three issues have been addressed over the decades, usually in an incremental, band-aid fashion. Peeling those band-aids back would require asking a different kind of question – for example, whether the issues would evaporate entirely if we stopped sharing mutable memory between threads. LLMs are trained on software as it exists today, so they produce stock answers. To explore alternatives, we would need to pose fundamentally different questions. That is outside the scope of this tool, but it is a reminder that the tool surfaces useful depth – it does not guarantee unconventional insight.

Conclusion

Is the Five-Whys Tool Working?

Yes. It produces progressively more detailed and actionable answers from a vague initial question. Further bench-testing across different questions would be worthwhile, but the core mechanism is sound.

What Has FDD Taught Us?

The iterative FDD spiral – sketching top-down, building bottom-up, and alternating between the two – is explicit and trackable. Each pass through the spiral either nails down a concrete part or reveals a gap that guides the next pass. There is no vague hand-wave of “iterative development”; every iteration has a clear before-and-after.

Parts behave like LEGO bricks. Once a part is built and tested, it goes on the palette and we stop thinking about it. The LLM part is a good example: wrapping a complicated block of external Go code behind a single shell-out Leaf took one short session, and after that the part was used multiple times throughout the design without modification or further thought. At no point did the existence of Go code pull us out of the design mindset and into language-level implementation details. The parts compose; the internals stay hidden.

Probes turn bench-testing into a first-class activity rather than an afterthought. We insert a probe, run the system, observe, and pull the probe out when we are done – exactly as an electronics engineer uses an oscilloscope probe on a live circuit. The fan-out mechanism makes this effortless: tapping a signal requires drawing one extra arrow in the diagram, not restructuring any code. The probe outputs in this example made the feedback-loop behaviour immediately readable, and removing them once they had served their purpose cost nothing.

Finally, production concerns – speed, cost, and memory footprint – have been deliberately set aside throughout. The design is clean enough and the parts are independent enough that a production-engineering team could take this design and optimise it without touching the architecture. Separating design-time thinking from production-time optimisation is itself a lesson: trying to do both at once tends to contaminate the design with premature constraints.

Parts Based Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Suggested Starting Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Articles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Videos

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Everything is a Black Box Part

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Drafting Rules For Node-and-Wire Diagrammatic Languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Building Multi-Language Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

@make

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Appendix A - How We Got Here

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.

Historical Choices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/failuredrivendevelopment.