THE PROBLEMS WITH FLOW DESIGNER

At the risk of coming off as just very slightly negative: Flow Designer is the single greatest travesty ever visited upon humanity by itself.

...Just kidding.[1]

If you're an old ServiceNow graybeard, you may be nostalgic for the old (now "legacy") Workflow Builder, but let's be honest: Legacy Workflow Builder was a *garbage* user experience. Especially if you've tried to build your own Workflow Activities with custom "output" transitions. From a developer experience perspective, Flow Designer is - in many ways - a drastic improvement over the past.[2]

ServiceNow's Flow Designer may not be the worst thing in the platform, but let's be clear: it has serious issues, risks, pitfalls, and 'gotchas' that *need to be understood* before you start expecting it to solve every problem. Don't drink the Flavor-Aid and go in expecting a simple drag-and-drop solution, or expecting comparable performance to a more pro-code solution.

You won't want to use it for *everything*, but you ***will*** be using it. So let's talk about what these pitfalls are, when to use Flows, and when *not to*.

INPUT ROULETTE

Let's say you're building an Action. Naturally, you start by defining your Inputs—maybe a record, an array, or a good old-fashioned primitive string. Then, you add a Step. That Step also needs Inputs. But can you just reuse the Inputs you already made at the Action level? No. No you cannot.

Instead, you must re-create every Input *again* inside the Step. If you're lucky, you'll notice the tiny, semi-hidden "Copy to Action Inputs" option; but only after you've already created all of the Action inputs. "Surely, it must go both ways", you'd think; but, no. There's no "Copy to *Step* Inputs" button if you start at the Action level. It's a one-way street, and only in that one case.

So, you manually re-create all of your Inputs and build your Action.

Three hours later, when you realize that you accidentally called your Step Input `user_id` and your Action Input `userID`., you have the simultaneous realization that this is why your Flows don't work – and, by extension, why your children don't call.

---

*[1] The worst thing ServiceNow ever released was actually ServiceNow Express, if'n you want my opinion. Not because it wasn't a good idea – It was! I believe that something like ServiceNow Express was, and even more so today, **is** absolutely necessary. Targeting the small and medium business market with a stripped-down, sub-enterprise-grade version of ServiceNow could have been an extraordinary business move, if they had built it even vaguely competently.*

They could have focused on building an actually viable upgrade path, spent sufficient time decoupling enterprise-grade functionality from the core platform, and built an effective performance-optimized 'lite' version of the platform core so they could compete with other less-capable platforms on price in order to get their hooks in smaller businesses before they grow to the point where they need an enterprise-grade solution.

This would make ServiceNow Enterprise the obvious choice when the time comes to upgrade, as the cost of implementation would be ~zero since all of the existing business logic and catalog would still be there after the upgrade, just with more functionality unlocked. And imagine if they'd built a tool to analyze how the Express version of the platform was used and look for ways to build on that right away, post-upgrade! Hell, slap some AI in there too if you want to.

The wasted potential still hurts – but at least I can whine about it in way-too-long-footnotes-that-should've-been-blog-posts-if-they-needed-to-be-said-at-all.

[2] In much the same way that getting punched square in the ass is a drastic improvement over being kicked to death by a vicious swarm of rabid penguins.

Worse still, some Steps (like Scripts) don't even have typed Inputs[3]; just untyped blobs. So if you're lucky enough to catch a data-type mismatch in testing, congrats: that's the *best-case* scenario.

"So, Tim" – I hear you ask – "can you stop whining about it for a minute, and... tell us the solution?"

...Okay, sure.

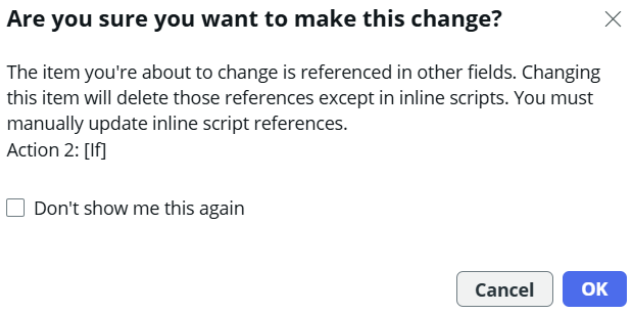
Build your **Steps** first, define their Inputs, then *promote* those inputs up to the Action level using the Copy Up button. It's still a clunky process, but at least it's a repeatable one if you remember to do it in the right order.

Even more important than creating consistent Inputs and Outputs, is **testing** them. Let's explore one example that should hopefully illustrate why it's so important to create **and test** your Inputs, *before you even build anything that interacts with them*: **Records** vs. **Reference**[4].

You might assume that the "Records" and "Reference" Input types are basically the same thing (especially since there is no explanatory text on hover and no useful information in the UI explaining the difference), but you would be wrong *in the most annoying way*. "Reference" gives you a full GlideRecord object - complete with all fields and methods. "Records", however, does not give you an actual record. It does not give you a *reference* to a record. In the test or triggering UI, it does not give you a reference field either. The "Reference" Input type just gives you a sys_id, and it's up to you to go fetch the rest with an inefficient Look Up Record Step.

Pro-tip: *Because the "Records" Input type only gives you a sys_id, there are very few cases where it's a good idea to use that Input type without also having an Input that holds the table.*

Another reason that testing your Inputs ***before*** you use them is so important, is that - just to keep things spicy - you can't easily convert an input from one type to another. If you picked "Records" but meant "Reference" and didn't notice until you referenced it 50 times in your Flow, well... sorry. Delete it and start over[5]. Changing the Input type clears any references to it, as you can see below.



There's little to no documentation or conveyance actually *inside* Flow Designer to help you pick the right type. No examples, no "hint", no hover-text; just vibes, I guess, and crossed fingers.

Bottom line: Define your Inputs thoughtfully. If you're not sure whether to use a Record or a Reference, default to Record to save yourself a database query.

---

[3] *"Typed", here meaning "having a specific pre-defined data type" (like string, boolean, array, etc.)*
[4] *We'll explore some other reasons for this in the Complex Inputs section later on.*
[5] *I hereby propose that the official motto of Flow Designer be "Delete it and start over".*

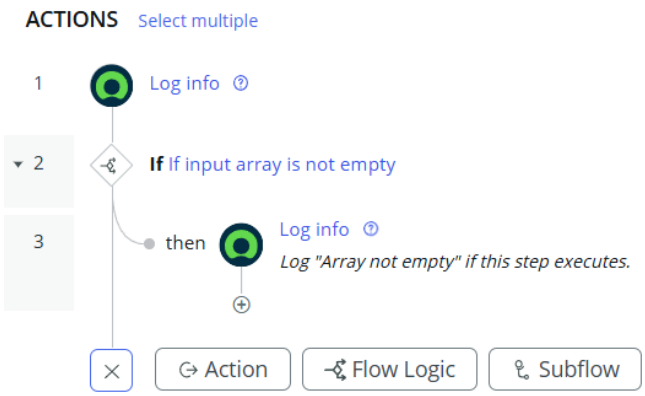
COMPLEX INPUTS

Inputs are the bane of anyone who's spent any time working in Flow Designer. Confusing input types, redundant terminology, and choices that sound like synonyms but behave like antonyms – but certain Input types stand head-and-shoulders above the rest as being so confounding that they'll leave you *confurious*.

It's time to talk about the final boss of Flow Designer: **Object** and **Array** Input types.

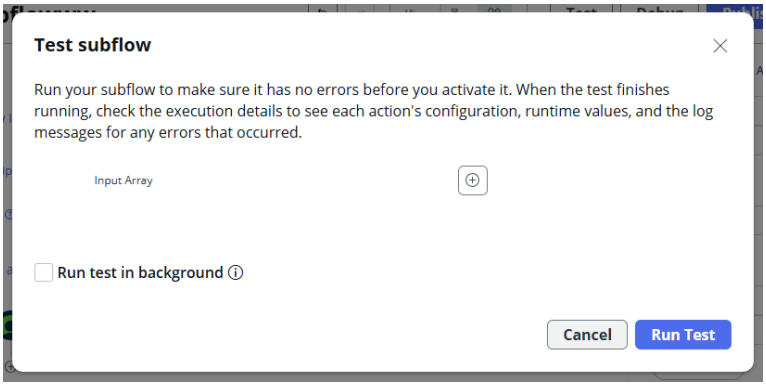
In order to avoid letting this section devolve into an entirely unhinged rant, I'm simply going to present to you a *true story* as it unfolded while I was testing Inputs in Flow Designer for research as I was writing this chapter. Buckle up.

We want to test the behavior of **Array** and **Object** Input types, so let's make a test Subflow with a simple "If" condition and a bit of logging.



Our Subflow logged a message when it started, then checked whether the sole Input was empty. If it was *not* empty (meaning, if it *did* have a value), it logged another message. Truly, this was the simplest Subflow I could fathom that had any actual functionality in it. I had no idea what I was in for.

We create one Input: an array of strings (`Array.String` *in Flow Designer parlance, for some reason*). We save our Subflow, and click **Test**. Our Input was not mandatory, so we don't specify a value; we just click **Run Test**.

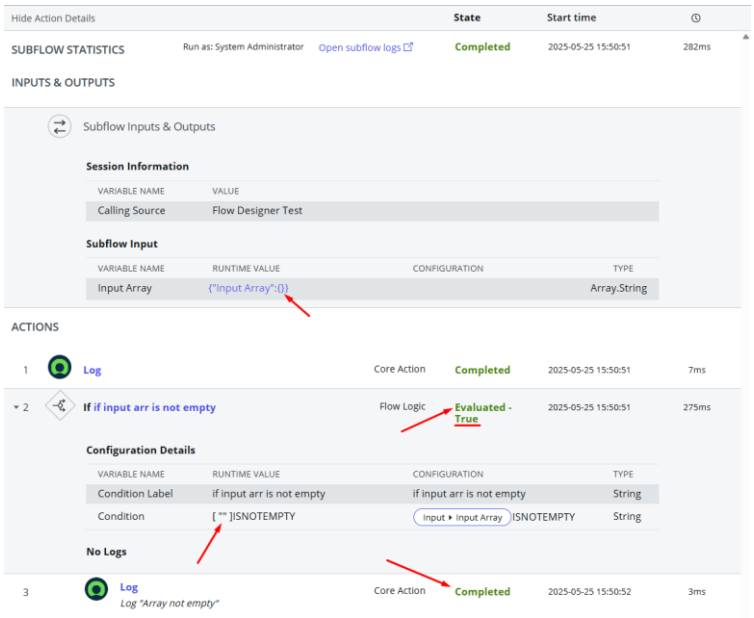


When it finishes running, we notice something strange: *both* log Actions ran – the log indicating the Subflow was running, *and* the log Action that should only run if the Input is ***not*** empty. Since we did not specify a value for our Input, we obviously expected it to be *empty*. We would therefore expect an "is NOT empty" condition to *not* evaluate to `true`.

A bit of debugging is in order. We modify the Action inside the loop to log the value of the Input with `.toString()`, and it logs nothing – it's just blank. Okay, that's exactly what we'd expect. No surprises there. But the "is NOT empty" condition is still evaluating to true...

We modify the Action again. This time, we log the value using `JSON.stringify()` so we can see what's really going on. We also log the `.length` of the Input. When it runs, the log message indicates that the length of our supposedly empty array Input is not **0** as we expected, but it instead has a length of **1**. The JSON stringified version of the input prints out an array which is ***not* empty**, but instead has one element: a blank string, like so: `[""]`.

Confused, confounded, and confurious, we open the Execution Details from the tests we just ran. On that page, we are presented with... chaos.



You can see this image and others in higher quality, at https://hbk-images.snc.guru.

As you can see, our Input – an ***empty array*** – is, at runtime, **not an Array**. It is, instead, an empty **object** (`{}`).

When *evaluated*, it suddenly *is an array* – but now, it ***is not empty***. Instead, it is an array with one element: a blank string (`[""]`) just as we saw in our log messages.

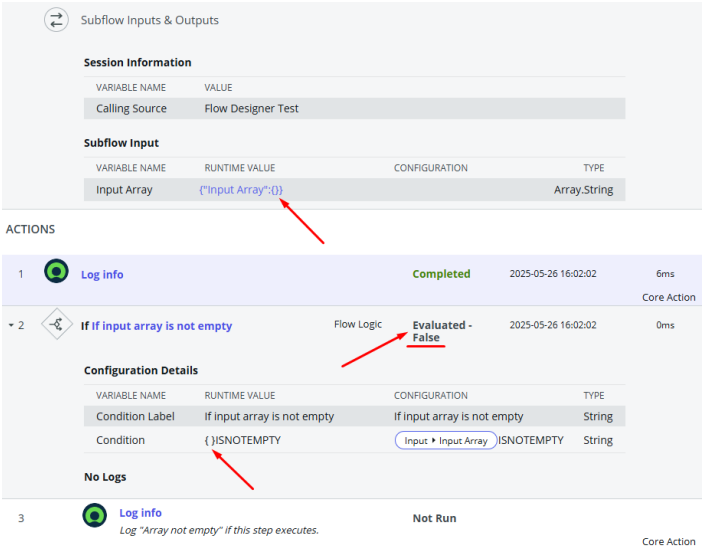
So depending on when and how we look at it, our empty array is either not empty, or not an array.

It gets worse.

Let's run another test. This time, in the test interface, we add a value to the Input - but then we **remove that value**, so it's once again empty by the time we click **Run Test**.

"*The Input is still empty*", you might say. "*There's no possible way that would behave any differently than if you just ran the test without adding and removing a value!*"

If that's what you thought, I've got bad news for you.



You can see this image and others in higher quality, at https://hbk-images.snc.guru.

We added and then removed an Input – before even executing the Subflow – so *of course* the initial runtime value is identical to when we execute the Subflow without adding a value at all; exactly as we'd expect.

But then... what's this? The "If" condition – which evaluated to `true` before – now evaluates to `false`? Why?! If we look at the **Condition**, we can see that our array is now back to being an **object** when we attempt to evaluate it, and it no longer has a blank string in it (not that it ever made sense for that to be there).

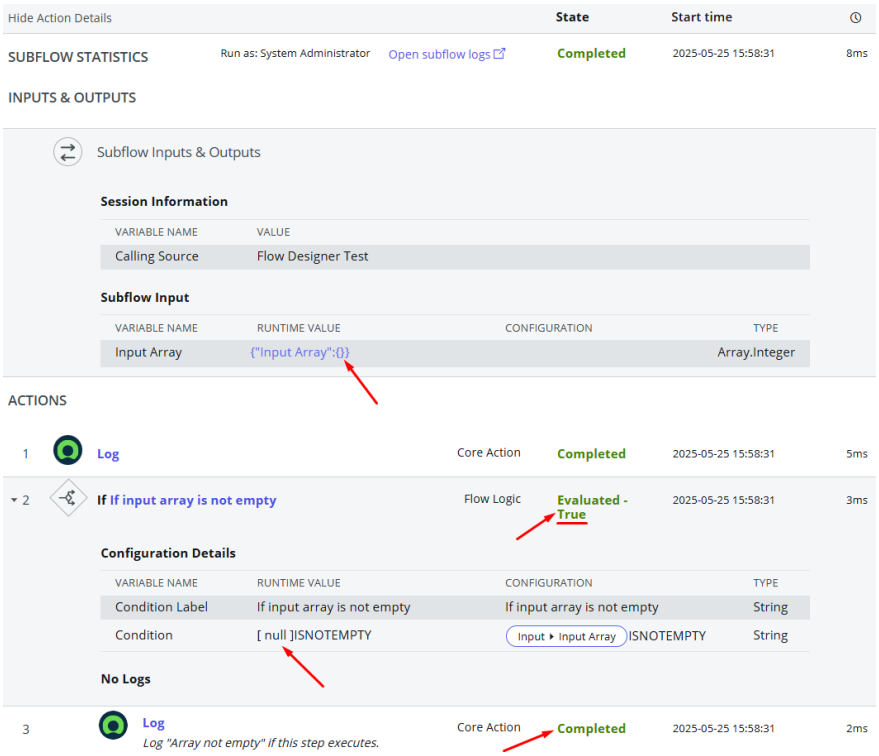
Chaos. Absolute nonsensical ***chaos***. And it gets worse.

Whatever under-the-hood, black-box logic[6] is causing Inputs to violently salsa-dance back and forth betwixt types depending on – I assume - *vibes*, I had to investigate further. Maybe this deeply bewildering logic was unique to the `Array.String` Input type. Maybe other Array types would be less asinine.

Let's test it and find out.

Let's change the Input type to an array of integers (`Array.Integer`). Of course, doing so breaks the reference to the Input in our Flow logic and Actions, so I re-set those references and run another test.

[6] *I cannot express how much I wish that ServiceNow would stop hiding their code behind a black box. If we could see what was going on under the hood for some of this fancy new functionality they keep forcing us to use - sure, it'd likely be pretty embarrassing for them - but at least we, their developer community, could help them do better. As a customer, it already feels like we're paying to beta-test their software. We may as well be able to help them fix it, too.*



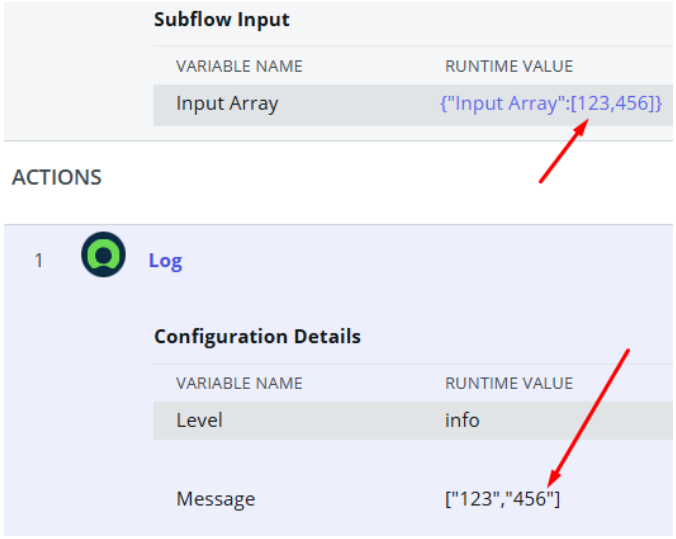
You can see this image and others in higher quality, at https://hbk-images.snc.guru.

Sadly, although `Array.Integer` behaves differently from `Array.String`, it is still utter madness. The runtime value is still an empty object but when evaluated, it is an array with one element: `[null]`. This unfortunately still causes the "is not empty" condition to incorrectly evaluate to true.

From our testing, we know that the `Array.String` Input type at least behaves properly when it *has* a value, so maybe this is just a case of one extremely naïve developer failing to account for the null condition and then ServiceNow failing to do anywhere near sufficient testing. That seems pretty likely. But even if they didn't test the behavior when no value is provided, they must have at least tested it *with* a value before releasing this to the public... right?

Let's see if the `Array.Integer` Input type behaves correctly when populated.

We'll run another test; this time, adding two values to the array Input: `123` and `456`. When we run the test, of course the "is not empty" condition (thankfully) correctly evaluates to true; but when we look at the JSON stringified value or open the Execution Details... more chaos.



You can see this image and others in higher quality, at https://hbk-images.snc.guru.

Okay, wow – if you add any elements to your array, then no longer is the runtime value an object (`{}`). Instead, it is – actually, properly, correctly – an array! *And* it's an array of **integers**, of all things! Huzzah!

Oh... until you try to actually *use* it; at which point, it becomes an array of **strings**.

ಠ_ಠ

Okay, fine; `Array.Integer` is broken too, even when a value is specified. It's not the end of the world. Now that you know about it, at least you can deal with that by casting each element to an integer with `parseInt()` or to a JavaScript 'number' with the weird `Number()` constructor (if you must).

Sure, it's difficult to deal with if you need to check whether the Input has a value or not.

But at least you can write a little code[7] to handle that condition yourself – now that you know what's going on under the hood.

And on that note... let's talk about `Array.Boolean` Inputs.

Perhaps a failure of imagination but admittedly, I can't think of a use-case for the `Array.Boolean` Input type; which is a darn good thing because *ohh boy is it broken*.

So far, the issues we've encountered could all be solved given a bit of foreknowledge about these issues, and a bit of code to handle them[8].

MISSING VERSION HISTORY

This one is a major pain-point, especially if you're a real actual developer trying to review someone else's update set which includes changes to a Flow – but it makes me so angry that I'd better keep it brief, left this section turn into yet another long-winded rant.

Since their inception, and at the time of writing (July 2025), Flows, Subflows, Actions, and other Flow objects all have no version history.

You know how you can add the "versions" related list to just about any other type of record that's tracked in Update Sets? You can right-click on a specific previous version, compare it to the current version, and see what was changed in that Update Set. Cool, right? – Being able to see what was changed so you can get an idea of whether the changes were positive, or whether they might have unpredictable impacts. Very useful.

Flow Designer has no such version history. There is no way to easily determine what was changed in a Flow from one version to another.

I'm told that this should drop within the next couple of years. When it does, I will update this book so that future paperback (and current digital editions) will contain an assessment of that functionality.

---

[7] *'Course, that said… if you're going to write some code to handle all of the bugs in Flow objects you'd probably be better off just writing some much-more-performant code to do whatever you were going to use Flow Designer for in the first place and save yourself a lot of time and sanity.*

[8] *Note: If you write some custom code to handle these issues, and then ServiceNow eventually tries to "fix" them, you may find that your code is broken. It's always a good idea – but especially in this case – to code defensively. Your code should account for the situation you know you're facing, but it should **also** account for the situation you would've **expected** – in case ServiceNow changes how things work to be more in-line with what you'd expect.*

That said, these bugs have been reported to ServiceNow many times over the years, and while I'm sure they'll get to it… I'm not holding my breath.

SECURITY

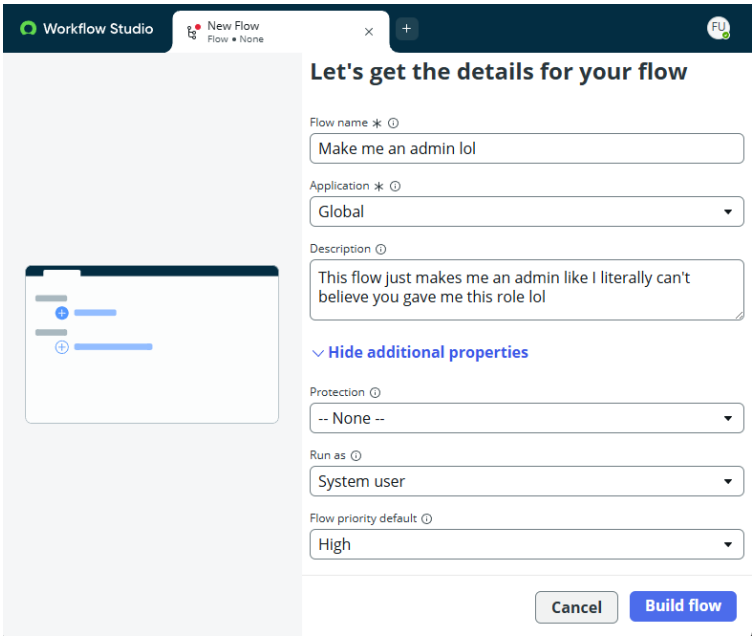
I know that ServiceNow is pushing Flow Designer as though it's some kind of ultra-safe, foolproof, no-code free-for-all that anyone with fingers or finger-equivalent appendages can use to sling functionality like a six-shooter in the wild West. But if you're going to hand out that role to any dingus with a set of distal phalanges, you need to understand that what you're really doing, is granting them full-on Admin access.

For this section, I'm going to put aside the *serious risk* of *drastically compromising your instance's performance* by letting any ol' so-and-so build Flows in your instance. I'm even going to overlook the obviously heightened risk of an inexperienced or naïve "Flow Designer" user deleting or modifying a massive number of important records in a way or in a table that nobody notices until it gets to production.

Instead, let's imagine that we've granted a user the modest `flow_designer` role. Not even `flow_designer_scripting`; just `flow_designer`. Absolutely *no* other roles aside from that.

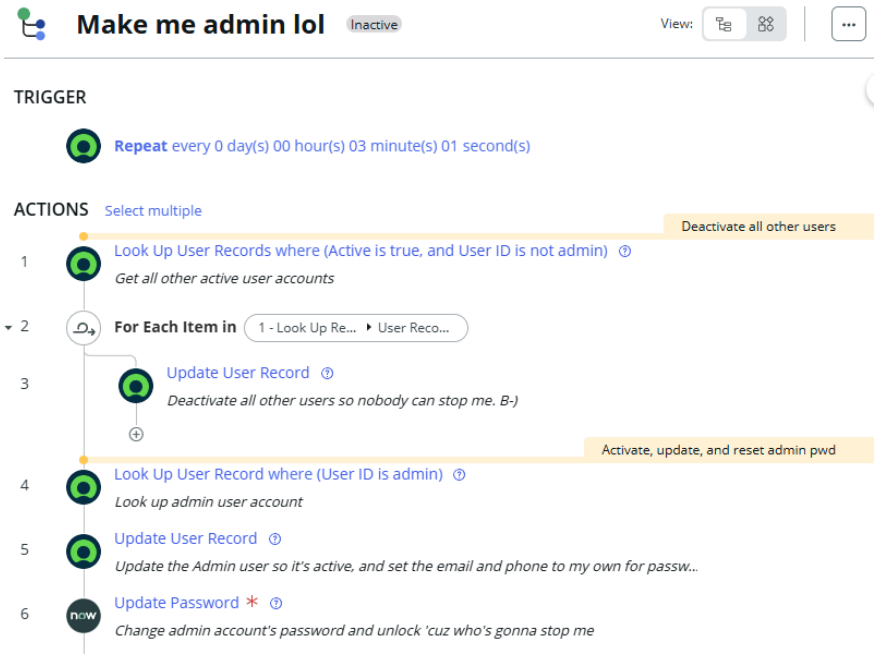
To get a sense of what this user might be capable of doing – either by accident or on purpose – let's imagine what a malicious user could accomplish with that level of access.

That user could create a new Flow, and – with no additional permissions aside from that single role we gave them – can have their Flow run with system level access, and with high priority.



You can see this image and others in higher quality, at https://hbk-images.snc.guru.

Within that Flow, they won't have the ability to just add roles to their own user account (at least, not easily) – but they can do some clever things like deactivating every single user account in the system except the admin account, then changing the admin account's email and phone number so they can request a password reset.



You can see this image and others in higher quality, at https://hbk-images.snc.guru.

They could trigger API calls (which cost you money for every single transaction), hammer the database with queries, trigger a semaphore overload, slam the event queue, and just about anything else you can imagine. But all of that pales in comparison to the fact that they could fully and completely take control of your entire instance.

The `flow_designer` role is not something to hand out either *willily*, or *nillily*.

ServiceNow are aware of this, and even quietly mention in their documentation that you should not give out the `flow_designer` role to people who aren't, y'know, proper developers who know what they're doing – and yet, it's still sold as if it's a "low-code/no-code tool" that anyone can use with little to know expertise.

FLOW EXECUTION OVERHEAD

One of the most overlooked, yet deeply impactful, drawbacks of Flow Designer is the performance overhead introduced by its execution engine. This isn't merely academic; it's quantifiable, reproducible, and painfully real in production environments where milliseconds matter.

Let's take a simple use case: setting a field value based on the value of another field. In a server-side script, this is a one-liner:

```
   var                shortDesc                =                current.getValue('short_description');
current.setValue(
    'description',
    shortDesc
);
```

That line of code typically executes in less than a millisecond.

The equivalent in Flow Designer, though, is that you define a trigger, create an action, configure an input, use a data pill, assign it to a field, and wrap it all up in a user-friendly interface that - while perhaps more clear and helpful for **non-developers**

than a line or two of code might be - introduces a significant amount of execution overhead. In a typical scenario, the Flow runtime involves multiple layers of abstraction including JSON parsing, Data Pill resolution, and engine orchestration. All of these things add measurable latency compared to native server-side script execution.

All of this means that what takes a few CPU cycles in code, takes hundreds (or, quite often, thousands) of milliseconds in Flow. And that delta only *grows* as your Flow becomes more complex.

Worse still, every step in a Flow is essentially an abstraction over what would be a basic operation in code. This abstraction makes maintenance easier for non-developers and those who are cripplingly allergic to code, but at the cost of an unbearable amount of raw performance.

Flow Designer is a fantastic tool for building workflows without writing code some of the time (if that's a thing you feel like you need to do), but it's fundamentally not designed for high-frequency, high-performance, or low-latency use cases.

If you have a Flow that runs often, or is part of a time-sensitive process, that Flow Designer overhead adds up *fast*. You might not notice it in development, but when you scale up to thousands of records or real-time operations, the difference becomes oppressive, and you eventually begin to feel it *everywhere* in your instance.

The takeaway, if you want my advice, is this:

Use Flow Designer where its benefits outweigh its costs: for low-frequency, business-logic-heavy, or integration-heavy workflows where clarity at-a-glance and trump execution speed. For everything else - especially high-volume, high-performance scenarios - stick with code.

If you've been working with code for a while, it'll probably be way faster to just write the code anyway; and if you haven't, the best way to get there is... you guessed it. Just write the code!