FLOW-LIKE

ONE PROGRAM · TWO HONEST VIEWS

FlowBook

The FlowScript Book

Build reliable software in code and as a visible workflow.

EDITION
Open edition · 2026

PUBLISHER
Flow-Like

WEB
book.flow-like.com

OPEN EDITION · 2026

Contents

INTRODUCTION

One Program, Two Ways to See It

Learn why Flow-Like FlowScript unifies typed code and a visual node graph, how the same workflow executes, and where AI-assisted software fits.

Most software has two lives.

There is the life its authors see: source files, services, packages, deployment definitions, dashboards, tickets, and the history of decisions that made the system what it is. Then there is the life everybody else sees: an interface that works—until it does not.

The distance between those lives is where expensive mysteries accumulate.

A process can be obvious to the person who built it and opaque to the person responsible for it at two in the morning. A deployment can be considered routine by the platform team and impossible to reproduce by the domain expert who needs one small change. An AI system can produce thousands of lines of plausible code in an afternoon while leaving nobody able to explain which operation failed, what it was allowed to access, or whether it will survive its first real workload.

FlowScript begins with a different expectation:

> *Running software should carry enough structure and evidence to explain itself.*

That does not mean every system can narrate its own correctness. It does not remove the need for engineers, operators, security reviewers, or careful design. It means the logic should not disappear when it leaves its author's editor. Its building blocks, boundaries, execution order, permissions, versions, and failures should remain inspectable in the environment where the software is built and run.

The graph and the sentence

Visual programming is often presented as a beginner's substitute for "real code." Textual programming is often presented as the inevitable destination once a project becomes serious. Both descriptions miss something important.

A graph can make relationships visible that source code asks us to reconstruct mentally. It can show where data comes from, which operation follows which, where execution branches, and which building block failed. That is valuable to a beginner, but it is not merely beginner friendly. During an incident, a good map is useful even to the most experienced engineer in the room.

Text has a different strength. It is dense, searchable, reviewable, and fast to manipulate. It scales better when a Flow contains many operations or when a developer needs to make a precise change across a large system. It works naturally with language tools and with the coding agents that increasingly participate in software development.

FlowScript refuses the choice between them.

FlowScript is Flow-Like's typed textual language for authoring Flows. It gives the same program a compact code view and an editable visual workflow while keeping execution inside the platform's governed node model.

In Flow-Like, Studio and FlowScript are equal authoring surfaces over one underlying Flow model. You can change the graph and read the resulting source. You can change the source, preview the resulting graph operations, and apply them. Neither view is decorative. Neither is an exported diagram that begins drifting away the moment it is created.

THE PRECISE TECHNICAL CONTRACT

FlowScript is an authoring language, not a second execution engine. Today, its text is parsed and reconciled into the persisted Board behind a Flow. The Rust runtime executes that graph. The two views have equal authoring authority; they are not two independently stored programs.

Why invent a language?

A visual platform that can solve only simple problems eventually forces its users to leave. To keep real application logic inside the platform, it must offer low-level operations as well as convenient high-level ones. That flexibility has a cost: honest graphs become large. A configuration format may preserve such a graph, but it does not necessarily make the graph pleasant to author or review.

Many workflow products answer this pressure by placing an arbitrary JavaScript or Python editor inside a node. It is an effective escape hatch. It is also where the visual model can stop telling the truth.

The node may appear as one tidy rectangle while containing a small application of its own. Its dependencies, network behavior, side effects, error paths, and assumptions are hidden behind the label on the box. Repeat this pattern across a workflow and the result combines the maintenance problems of conventional software with the weaker navigation of a visual configuration. The graph looks simple because the complexity has been concealed, not removed.

FlowScript exists to provide textual scale without creating that hidden channel. Its familiar syntax still resolves to typed nodes, pins, execution paths, variables, functions, Events, and other concepts the graph can represent. When the existing catalog cannot express a reusable capability, the extension boundary is a package containing scoped WebAssembly nodes, not an invisible script pasted into the middle of an application.

This is constrained freedom. The outcome can be broad; the way we reach it is deliberately more opinionated.

The system around the language

FlowScript is part of Flow-Like, not a language floating apart from its runtime.

An App supplies the project and governance boundary. Flows contain its executable logic. Boards are the persisted graphs behind those Flows. Events expose typed entry points to an API, schedule, page, chat, quick action, or another supported surface. Data Studio keeps files, tables, queries, and ontologies close to the application. Packages add reviewed and versioned capabilities. The runtime executes the resulting graph and records evidence about the run.

This wider boundary matters because a language alone cannot make software reliable. Types do not configure production storage. A readable function does not establish who may invoke it. A clean graph does not create versioning, logging, auditing, deployment, or rollback. If every team must reconstruct those disciplines around every new application, the domain problem again becomes the smallest part of the work.

Flow-Like's thesis is that much of this surrounding discipline can be supplied once, through the platform, and reused by every App. Throughout this book we will distinguish that thesis from the exact guarantees of a particular release. Security, performance, scale, cost, and deployment claims will be tied to implementation evidence or measurements rather than treated as adjectives.

AI changes what is cheap

Flow-Like was not conceived as a response to generative AI. The major-incident call that supplied its founding question came earlier. So did the influence of visual programming and the conviction that domain experts should be able to work directly with application logic. The role of AI crystallized later.

AI did not replace the original argument. It made the fault line wider.

> *AI made scaling development cheap. Maintenance and architecture skill remain scarce.*

Here, *scaling development* means multiplying the amount of software we can produce. It does not mean that the resulting software scales.

AI-assisted development is a real advance. Vibe coding shows how dramatically it can lower the cost of a first prototype. An experienced developer can explore an unfamiliar library, remove repetitive work, test several approaches, and turn a clear design into a working implementation much faster than before. A founder or domain expert can reach a prototype without waiting for a complete delivery team. Those gains are worth keeping.

But generation compresses the time required to emit code. It does not automatically supply a sound architecture, discover an unstated operational requirement, or recognize that a plausible local decision will become disastrous at production scale. An experienced engineer often notices the model's false assumption because it conflicts with a mental model built over years. To somebody without that context, the same answer may simply look finished.

The result can be a prototype that technically exists but is expensive to run, difficult to deploy, and almost impossible to maintain. More code was produced. The scarce work merely moved downstream.

This is not an argument that only expert programmers should use AI. It is an argument for putting more of their hard-won knowledge into the environment in which AI works: typed contracts, approved operations, bounded changes, visible structure, shared runtime services, and evidence from actual executions.

Do not ask a model to add one

In August 2025, an experimental Python package called `vibeincrement` appeared with one job: increment a positive integer. Instead of evaluating `n + 1`, its implementation serializes the number and the requested oper-

ation, sends them to `gpt-4o-mini`, asks for a structured integer response, and returns the model's answer. The project explicitly warns that it is not for production.

The package reads as satire, but it is an experiment—not evidence that production teams routinely add one this way. The example works because it exaggerates a genuine temptation: once a model is available, every problem starts to look like a prompt.

That is the wrong default.

If an operation can be expressed deterministically, express it deterministically. Arithmetic, parsing, exact filtering, known routing rules, identifiers, schema validation, and ordinary data transformations do not become better merely because a model performs them. In FlowScript, an increment should remain exactly what it says:

```
let nextRevision = revision + 1
```

Every unnecessary model call adds a probabilistic boundary. It may also add network latency, variable cost, provider dependence, new data exposure, and another failure mode. Even a local model consumes compute and makes an exact operation less exact.

Flow-Like's AI authoring assistant, FlowPilot, encodes this rule directly in its current guidance: model calls are for semantic work, not arithmetic, counting, number parsing, or revision increments. The larger doctrine is just as blunt:

> *If we can solve it deterministically, we should. Use a model only where uncertainty is useful.*

Two very different roles for AI

The phrase *AI software* hides an important distinction.

First, AI can be an **author**. It can propose FlowScript, assemble a Flow, connect existing nodes, or adapt a known App. In that role, its output should be treated as an untrusted change, not as a privileged command. The catalog limits what it can call. Declarations tell it the real signatures. Types and reconciliation reject invalid connections. A preview can expose the graph operations before they are applied. The current reconciler also rejects an edit that would leave any one layer with more than one hundred nodes, pushing large logic into named, reviewable functions.

Second, AI can be an **operation inside the running application**. A model may classify an ambiguous request, extract meaning from prose, summarize a document, or formulate a response. That operation is intentionally probabilistic. It therefore belongs at an explicit node with typed inputs and outputs, a narrow permission surface, recorded usage, an error path, and deterministic logic around it.

These roles need different controls, but they share one principle: the model does not get to erase the structure of the application.

For critical work, “correct 99 percent of the time” can still be unusable. It means roughly one miss in every hundred opportunities before we account for the way failures accumulate across steps and repeated executions. The right response is not to pretend a model has become deterministic. It is to make the uncertain boundary small, evaluate it for the actual task, handle its failure explicitly, and keep the rest of the Flow exact.

Spend intelligence where it matters

The most capable model is not the correct default for every call. The target is the smallest model that satisfies the measured requirements of one focused task.

Deterministic nodes should prepare the input before the call: retrieve only the relevant material, remove fields the model does not need, and compute exact values outside the prompt. The model should receive a narrow question and return a narrow, typed result. Deterministic nodes should then validate, route, store, or reject that result. This reduces cost and latency, but it also reduces the number of places in which a plausible mistake can enter the system.

Cost control follows the same hierarchy. We should be able to attribute model use to an App and a user, see which provider and model were invoked, set limits, and select models under explicit cost and capability preferences. Over time, each App should carry task-specific evaluations so a model earns its place through observed quality rather than reputation or size.

WHAT EXISTS TODAY

Flow-Like's hosted-call accounting can record App, user, model, provider, token, latency, and cost context for model and embedding usage. It exposes App-wide and technical-user usage limits, and user-owned profiles can select hosted or local models using configured weights such as cost, speed, reasoning, safety, and coding. A feature-gated governance inventory can reconcile models declared in an App with models observed in use.

That is not yet the whole doctrine. Editing a separate budget for every human user, evaluating model quality automatically for each task in an App, and guaranteeing selection of the smallest adequate model remain incomplete. Provider-reported cost can also arrive only after a hosted call, so a hard limit is not a universal pre-call spending guarantee. Later chapters will keep those boundaries visible.

Reuse is another form of efficiency. Many applications are not blank-page inventions; they are variations of a shape that already works. A governed document assistant, for example, should be forked, configured with the new App's data and permissions, and adapted where the domain genuinely differs. Starting from a visible, parameterized App gives a human or an AI a smaller solution space and the organization a lineage it can inspect.

This is Flow-Like's AI bet: use AI to expand who can build software without expanding the software's blast radius. Let it accelerate experienced developers. Let it help domain experts express what they know. Let it perform the semantic work for which uncertainty is actually valuable. But keep the application's spine typed, visible, deterministic, and cheap.

The useful question is therefore not, "Where can we insert AI?" It is:

> *Where is uncertainty necessary, and how small can we make that boundary?*

Who belongs here

This book is for people who arrive with different kinds of expertise.

You may be an enterprise developer who wants a faster path from a domain requirement to a maintainable service. You may be a founder or technical business user who understands the problem deeply but does not want to become an expert in ten adjacent infrastructure disciplines. You may be a student learning how data and execution move through a system. You may be an experienced Rust developer building WASM nodes for everybody else. You may be working with an AI coding agent and need the result to remain understandable after the chat window closes.

The shared requirement is not the same programming background. It is the need to collaborate around logic that remains visible.

The early chapters keep the graph and source side by side. Later sections open the machinery for readers who want to understand parsing, reconciliation, execution, packages, governance, and deployment. You can skip those internals and return to the main application path without losing the thread.

How we will build

We will not learn FlowScript as a disconnected sequence of syntax features.

First, we will enter the incident that supplied the founding question. Then we will state the design principles clearly enough that you can challenge them. In Chapter 4 we will build Incident Triage: a small Flow with no account, model, API key, or external dependency. You will see the result before learning every construct. You will edit it from both views and break it on purpose.

From there, each language feature will answer four questions:

1. What problem does it solve?
2. How does it read in FlowScript?
3. What graph does it represent?
4. What evidence does it leave when it runs or fails?

The capstone, Incident Room, will then assemble those ideas into a complete private document assistant. It will ingest runbooks, retrieve relevant material, expose narrow tools, answer through an application interface, and let us inspect the path of a real execution. Later we will extend the catalog with a WASM node and follow the App into packaging, governance, compiled artifacts, and deployment.

We will take constraints seriously enough to show their edges. If a language construct does not round-trip yet, the book will say so. If an execution boundary depends on deployment policy, the book will name that dependency. If a benchmark describes one narrow workload, it will not be inflated into a universal performance claim.

That candor is part of the premise. Software cannot explain itself if its documentation hides the parts that are still being built.

The first question, then, is not about syntax. It is the question from a major-incident call in the middle of the night:

Why did everybody have to wait for the one person who understood the system?

PART I

Software That Explains Itself

Begin with the founding incident, establish the operating principles, and build the first Flow in two honest views.

PART I · CHAPTER 01

The 3 A.M. Call

A major incident reveals why Flow-Like was built for visible software structure, operational evidence, and domain knowledge that survives its author.

At three in the morning, every minute feels expensive.

On this particular night, the feeling was literal. I was part of a major-incident call inside a large enterprise. Something in an important chain of systems had failed, and every hour of downtime risked millions in damage.

Nothing technical was visible to us. The only concrete signal came from the domain experts: production was on hold.

I did not know the systems involved. I did not know their interfaces, their dependencies, or the decisions that had shaped them. That alone was not unusual. No one in a large organization understands every system. The reason for an incident call is to bring the necessary knowledge into one place.

But the knowledge we needed was not in the room.

It was not in the system, either.

1.1 Waiting for the one person

We were waiting for the system expert—the person who understood the affected system well enough to tell us what we were looking at.

Except there was almost nothing to look at. The domain experts could describe the consequence: production had stopped. They could not point us to the operation, interface, or component that had caused it. The incident was undeniable, but its technical path was invisible.

Until the system expert could be reached, the rest of us could do little more than inspect fragments, exchange partial theories, and wait. The system was important enough that its failure could cause enormous damage. Yet understanding that system still depended on locating a particular human being in the middle of the night.

Hours passed.

That was the part I could not reconcile. Complex systems will fail. Interfaces will behave in unexpected ways. Dependencies will become unavailable. Data will arrive in a form nobody anticipated. Reliability cannot mean pretending those things will never happen.

But why did failure also have to mean blindness?

The organization had engineers, operators, processes, and escalation paths. The call itself existed because people took reliability seriously. Nevertheless, the system could not reveal enough about itself for an unfamiliar responder to follow the execution, identify the responsible operation, and understand what had gone wrong.

The scarce resource was not computing power. It was context.

The system expert knew which component mattered, what a particular interface expected, and which behavior was normal or evidence of the fault. That context was what eventually allowed the failure to be identified. But the knowledge lived in a person, not in a form that traveled with the running software.

The incident therefore had two failures. The first was the technical failure that caused the outage. The second was the system's inability to explain the first.

1.2 The documentation was somewhere else

This is usually where a discussion about enterprise software turns toward documentation.

Perhaps there should have been a better diagram. Perhaps a runbook needed another section. Perhaps a ticket should have linked to an architectural decision, or an interface should have had more complete

API documentation.

All of those things can help. None of them fully resolves the underlying problem.

A diagram describes the system at the time the diagram was drawn. A runbook describes the failures its authors anticipated. A ticket records a piece of work from the perspective of the people who performed it. Each artifact can be correct when created and still drift away from the software that eventually executes.

The more places the explanation can live, the more opportunities there are for the explanation and the execution to diverge.

This is not necessarily negligence. Maintaining two representations of the same thing is difficult. Maintaining five is worse. Under delivery pressure, people change the artifact that makes the system work. Updating every secondary explanation becomes a separate task, dependent on time, discipline, and memory.

The code may remain technically accurate, but code alone does not necessarily explain the running system to an operator, a domain expert, or a developer arriving during an incident. Configuration may be scattered across services. Runtime behavior may depend on infrastructure that belongs to another team. Logs may report an error without placing it inside the larger process. The architecture exists, but it has to be reconstructed.

Documentation is then treated as a destination somewhere outside the system: a page to find, a repository to search, or a person to call.

What I wanted was different. I wanted the executing system to carry its explanation with it.

That does not mean generating a paragraph of prose from source code and calling the result self-documentation. It means making the actual structure of the program visible. It means exposing its operations and boundaries, preserving the relationship between those operations and their runtime evidence, and showing a responder where execution stopped making sense.

The documentation should not merely describe the system. It should remain connected to the thing being described.

1.3 What the failing block should have told us

Imagine the same incident with the application represented as a visible Flow.

The responder can see the operations that make up the process and the connections between them. Inputs and outputs have types. External calls are identifiable as operations rather than disappearing inside arbitrary code. Execution evidence belongs to the individual building blocks that produced it.

One node shows the failure.

That node does not magically solve the incident. A graph cannot repair an unavailable dependency or decide how a business process should behave when a critical input is missing. But it can change the first hour of the response.

Instead of beginning with *Who knows this system?*, the team can begin with *What happened at this operation?*

An unfamiliar responder can inspect the failing block, see what reached it, understand what it was supposed to produce, and trace the surrounding path. The visual structure provides the map. Typed boundaries narrow the possible explanations. Per-node logs connect runtime evidence to the operation that generated it.

The system begins the incident by giving the responder a place to stand.

The Flow-Like feature that most directly answers that night is tracing tied to individual nodes. The execution log identifies the building block involved in a failure, and from that log the responder can open the node's recorded result with one click. Instead of beginning with the whole application as an undifferentiated problem, the investigation begins at the block where the run failed.

The textual view matters too. A large program cannot depend on a canvas alone. Experienced developers need a representation they can navigate, compare, review, and edit efficiently. The point is not to replace code with boxes. It is to let the code and the visible workflow describe the same program.

During normal development, the text is often the fastest way to work. During debugging, tracing, onboarding, and conversation, the graph can make the same logic easier to follow. Neither view is a disposable export. A change made in either view changes the Flow.

Looking back, I believed that this kind of system would have made the incident understandable to someone outside the original team within minutes instead of leaving a room waiting for hours. That is a counterfactual; it cannot be proven against an event that already happened. It is also not a promise that visible software never fails.

The more defensible promise is this: when failure happens, the structure needed to investigate it should not be trapped in the memory of the original author.

Flow-Like does not begin with the fantasy of eliminating every incident. It begins with the demand that incidents become legible.

1.4 Domain knowledge is not the lesser skill

There was another problem hiding inside that call.

Enterprise software is often shaped by people who understand the business domain deeply but do not identify as specialist programmers. They understand the operation, its exceptions, the regulations surrounding it, and the consequences of getting it wrong. That knowledge is not secondary to software development. It is the reason the software exists.

Yet conventional development often places those people at a distance from the implementation. Domain experts describe their needs to one group, which translates them for another group, which turns them into code that becomes difficult for the original experts to inspect. Each translation can lose information.

The usual response is to give domain experts simpler tools. Too often, simpler means less expressive: a toy surface for small tasks, followed by an escape into conventional code when the work becomes serious.

That division creates its own trap. Once important behavior disappears into an opaque code block, the visual representation stops being an honest account of the application. The workflow shows that *something*

happens, while the real program lives behind a rectangle labelled with a reassuring name.

My background in game development suggested another possibility. Unreal Engine Blueprints demonstrated that a visual graph could be a genuine programming surface and an accessible route into programming. It did not have to be a picture pasted on top of unrelated code.

But enterprise workflows can become large, particularly when a platform exposes sufficiently low-level building blocks to solve real problems. Large graphs are difficult to author and review as graphs alone. Experienced developers need text. Domain experts, new programmers, operators, and people debugging unfamiliar systems benefit from the visible Flow.

The answer was not to choose one audience or one representation.

The answer was to let both groups work on the same program.

A developer can use text to manage a large implementation. A domain expert can inspect its process as a workflow. A beginner can enter through the visual model and gradually learn the textual one. During an incident, someone who did not write the program can follow the graph and connect runtime evidence to the responsible building block.

The visual view is not a lesser form of the code. Domain knowledge is not a lesser form of engineering knowledge. Each reveals something the other needs.

1.5 The question that became Flow-Like

I did not leave the incident with a grammar, a runtime architecture, or a finished product plan.

I left with a question:

How can a system important enough to cost millions when it fails reveal so little about itself?

That question later connected with the lessons from visual programming and with the reality of enterprise teams. It became larger than incident response.

Could the program's visible structure remain connected to the program that actually runs? Could text and workflow be equal ways of authoring the same logic? Could logs lead directly to the building block that produced them? Could domain experts understand the application without reducing what experienced developers were able to build?

And could the platform take responsibility for the disciplines that every serious application needs—execution, permissions, data, governance, and deployment—so that each team did not have to reconstruct them independently?

The answer did not arrive as a finished product idea during the call. The significance of the incident crystallized later, when it connected with visual programming, the needs of domain experts, and the recurring difficulty of operating enterprise software whose explanation lives outside the system.

These questions became Flow-Like.

FlowScript is one part of the answer: a textual way to author the same constrained, typed building blocks that appear in the visual workflow. Flow-Like is the surrounding platform that authors, runs, and observes those Flows.

The aim is not software without complexity. Serious domains are complex, and hiding that complexity behind an attractive interface does not remove it. The aim is software whose complexity is organized, inspectable, and connected to evidence.

It is software that explains itself—not perfectly, and not only in prose, but through the structure it executes.

The test is simple to state. When a Flow fails at three in the morning, a capable person who did not build it should be able to see where to begin.

They should not have to wait for the one person.

PART I · CHAPTER 02

The Manifesto: Constrained Freedom

Explore Flow-Like's principles for typed building blocks, legible workflows, safe extension, governed execution, and freedom without hidden liabilities.

The principles behind FlowScript begin with an uncomfortable trade.

We give up some unconstrained implementation freedom. In return, we want software that is easier to understand, safer to change, more consistent to run, and less dependent on the one person who knows how everything fits together.

That trade is deliberate. It does not mean that builders should be limited to trivial applications or forced into a small set of templates. It means the platform must provide enough useful building blocks that people can solve real problems without repeatedly escaping into invisible, ungoverned code.

Constraints are only valuable when they carry their own weight. They must prevent meaningful classes of mistakes while leaving the intended work practical. Otherwise, users will route around them—and they will be right to do so.

This is the standard against which every part of FlowScript and Flow-Like should be judged.

2.1 Software should be reliable and efficient

Reliability is often treated as work that begins after an application has been written.

First comes the feature. Later come deployment, authentication, logging, auditing, monitoring, recovery, permissions, secrets, scaling, and cost control. Each concern is assigned to another specialist, another tool, or another backlog. A prototype moves quickly because it temporarily ignores all the disciplines required to keep it alive.

But those disciplines are not outside the software. They determine whether the software can be trusted.

A program is not reliable merely because it produces the correct result on the author's machine. It must also behave predictably when inputs are unexpected, dependencies fail, permissions change, or the original author is unavailable. When it does fail, the people responsible for it need enough evidence to determine what happened. When it changes, reviewers need to understand what changed and what may be affected.

Reliability therefore spans the entire path from authoring to recovery:

- operations should have explicit inputs and outputs;
- incompatible connections should be rejected as early as practical;
- execution should leave useful evidence;
- failures should lead back to the responsible part of the Flow;
- sensitive values should not become ordinary source text;
- changes should pass through understandable, reviewable operations; and
- the execution environment should supply common operational concerns consistently.

Efficiency matters at two levels. Runtime efficiency matters because enterprise applications may execute often, process large workloads, or sit on critical paths. Human efficiency matters because rebuilding the same infrastructure and rediscovering the same failure modes consumes far more time than writing the domain logic itself.

Flow-Like's ambition is to provide a shared execution contract around every Flow. The exact behavior and operational maturity still depend on the release and deployment target, so this book will not pretend that every environment is identical. The principle is more durable than any

provider implementation: application builders should spend most of their effort on the problem they understand, not on reconstructing the surrounding disciplines for every new project.

2.2 A Flow should be organized and readable

Readable software is sometimes dismissed as a matter of style. In FlowScript, it is part of correctness.

Software rarely belongs to one person for its entire life. It is read by developers, domain experts, reviewers, operators, security teams, future maintainers, and increasingly by AI systems. Each reader approaches it with different knowledge. A developer may recognize a programming pattern immediately but not understand the business rule it implements. A domain expert may know that rule perfectly but struggle to recover it from framework code and infrastructure configuration.

A Flow should reduce that distance.

The visual view exposes operations and their relationships as nodes, pins, and wires. The textual view gives experienced developers a compact way to navigate and edit larger programs. Neither view should be treated as decorative documentation. Studio and FlowScript are equal authoring surfaces over one underlying Flow model.

That precision matters. FlowScript is not a second program that happens to resemble the graph, and the graph is not a diagram that developers must remember to update. In the current implementation, the Board is persisted. FlowScript is parsed and reconciled into changes to that Board, and the runtime executes the resulting graph.

The two surfaces serve different forms of attention. Text is good at density, search, repetition, and large changes. A graph is good at locality, paths, boundaries, and following a run through visible operations. A readable Flow lets a person choose the representation that best answers the question in front of them.

Organization is therefore not an aesthetic layer added after the logic works. It is how the logic remains available to the team. If a Flow can only be understood by collapsing it into a hidden block of general-purpose

code, the visual structure has stopped telling the truth.

2.3 Hard things should be fast to realize

Making software safer is easy if we make useful work unbearably slow. That is not the goal.

FlowScript should make difficult applications faster to realize by turning recurring engineering work into reusable, typed building blocks. The node catalog provides operations that authors can compose instead of implementing every capability from scratch. Flow-Like surrounds that logic with platform services for application structure, permissions, data, interfaces, execution, and other shared concerns.

The catalog is not a fixed list, and this book will not advertise a permanent node count. Its important property is the model: each node presents an inspectable contract. Inputs and outputs become pins. Connections can be checked against schemas. An operation in the source corresponds to a building block in the graph. Execution evidence can refer to that building block instead of only to a distant line in a combined log stream.

This does not make hard problems easy in the careless sense. Domain decisions remain hard. Data models remain consequential. External systems remain unreliable. Security and operating policy still require informed people. The platform cannot repeal those realities.

It can remove repetition that does not distinguish one domain problem from another.

A startup founder should not need to become an infrastructure specialist before testing a useful application. A domain expert should be able to express process knowledge without first learning every discipline of conventional software delivery. An experienced developer should be able to extend the available capabilities without forcing every later user to inherit the extension's internal complexity.

The measure is not how quickly we can produce a demo. It is how quickly we can reach something understandable enough to review, structured enough to extend, and disciplined enough to operate.

2.4 Why a language, not only a graph

Flow-Like began with visual workflows, but a serious workflow platform cannot stop at a handful of large, high-level boxes.

If builders must remain within the platform's constraints, the platform must be flexible enough to express their actual use cases. That requires low-level building blocks. Authors need to transform values, branch, iterate, coordinate operations, manage state, call services, and compose reusable pieces.

Low-level building blocks make the system honest, but they also make graphs larger.

At small scale, a graph can be wonderfully direct. As the number of operations grows, physical space becomes a cost. Related logic spreads across the canvas. Repeated patterns become tedious to author. Following wires across a large Board can be slower than reading a compact expression or function. A serialized configuration would not solve this problem; it would merely replace a spatial structure with an implementation format that humans were never meant to write.

One common response is to add a code node: an empty box containing arbitrary JavaScript, Python, or another general-purpose language. That makes the graph smaller, but only by hiding the application inside it.

The box may show one operation while internally performing dozens of calls, transformations, branches, and side effects. Its visible pins no longer describe the real structure. Debugging crosses from the workflow into an embedded program with a different set of tools and assumptions. Governance can inspect the box's label, but not necessarily the behavior concealed behind it. The workflow becomes a shell around fragments of unrelated application architecture.

We reject that escape hatch. It creates what we think of as a Frankenstein application: part workflow, part embedded script, with the maintenance costs of both and the transparency of neither.

FlowScript exists because a graph alone is not always the best authoring surface, but the alternative must preserve the graph's truth.

It expresses the same typed building blocks in text. Its imports, calls, values, branches, loops, functions, handlers, and other language structures represent logic that can still become visible workflow structure. Editing the text changes the underlying Flow model. Editing in Studio changes what the textual surface represents.

That is why FlowScript is a language rather than a configuration dump. It gives authors deliberate syntax for expressing and composing logic at a scale where text becomes more effective, while retaining a path back to nodes, pins, wires, and execution.

It is also why FlowScript is an authoring language, not a separate runtime. Its purpose is not to bypass the Board. Its purpose is to make the Board manageable without making it dishonest.

The two-way contract is still an evolving surface, and some constructs require careful handling or remain less convenient in one view than the other. We will name those edges rather than hiding them behind a claim of perfect parity. The principle is the destination: one program, two serious ways to work with it.

2.5 Dangerous things should be hard to do accidentally

No platform can make all harmful behavior impossible. The meaningful goal is narrower and more practical: common dangerous actions should require explicit boundaries, visible intent, or additional review.

Typed pins are one example. They move some mistakes from runtime surprises into authoring feedback. Types do not prove that a business decision is correct, but they can prevent connections that do not satisfy the operation's declared shape.

Secrets are another. FlowScript is designed so sensitive values do not need to become ordinary source literals. The current implementation includes redaction and protected deletion paths. Those mechanisms reduce risk, but an absolute claim that secrets can never appear in any preview, log, serialization, or error path would require verification across every path and release.

Extensions follow a similar philosophy. Custom nodes are packaged as WASM components with declared capabilities and a defined interface to the host. Interactive clients can present package and permission information before a run, and package publication can pass through review. These are meaningful layers. They are not a license to describe every node as completely isolated or automatically safe. Resource limits, network rules, ambient host access, remembered trust, and non-interactive executions must all be verified before the book presents them as guarantees.

Versioning, deletion guards, package access states, and publication gates are further examples of constraints that can carry their weight. Each makes an important transition more explicit. Yet each also has a precise boundary. A Flow version does not necessarily freeze every package, data source, configuration value, and external dependency. A public review does not automatically prove that a package is harmless. Governance evidence does not remove the need for operating policy and human judgment.

Good constraints tell us what they protect. Responsible documentation also tells us what they do not.

2.6 Freedom of outcome, opinions about implementation

FlowScript is opinionated about how software is assembled because Flow-Like aims to remain broad in what that software can accomplish.

Those positions are not opposites. A road network is useful because it constrains where vehicles travel while connecting many destinations. Typed nodes, declared capabilities, shared execution rules, and visible structure play a similar role. They narrow the ways behavior enters a Flow so the resulting system can be inspected and operated more consistently.

When a required capability does not exist, an experienced developer can build a custom WASM node and expose a clear contract to the rest of the team. The domain expert can then use that capability without inheriting its implementation language or internal complexity. The extension adds a building block instead of creating a secret application inside a box.

AI does not get an exception to these principles. As an author, it proposes reviewable changes over declared, typed operations. Inside a running App, it should occupy only the smallest boundary where ambiguity or judgment is genuinely required; exact work stays exact. These constraints do not guarantee correctness, but they make generated and model-assisted software easier for humans and tools to inspect.

We are willing to offer fewer arbitrary ways to inject credentials, deployment assumptions, invisible side effects, or unreviewed code. That is the freedom we trade.

The freedom we want to preserve is more valuable: the ability of a mixed team to solve its actual problem, to understand the result, to fork and adapt useful applications, and to keep operating them after the first author has moved on.

2.7 Never take the shortcut that worsens the product

The final principle is the hardest because it applies to us as much as it applies to the user.

We will not accept a shortcut when it makes the final product worse.

That does not mean every feature must arrive fully formed. FlowScript and Flow-Like are evolving systems. Authoring parity has edges. Client capabilities may differ. Deployment targets have different levels of operational maturity. Security boundaries require testing across more than the happiest execution path. Performance, scale, and cost claims require measurements rather than confidence.

The shortcut would be to smooth those differences into a perfect story.

The better product is built by stating the boundary, improving it, and giving users enough information to make a sound decision today. In this book, changing capabilities will be marked as current, preview, or vision. Claims about security, scale, compliance, cost, and deployment will be tied to implementation evidence or measurements. Unsupported edges will be taught as edges, not buried as surprises.

There is a second tension. If the robust path is consistently slower, more awkward, or less expressive than the escape hatch, people will choose the escape hatch. A manifesto cannot blame them for that. The platform must earn its constraints by making the disciplined path practical.

That is the promise behind constrained freedom: not less ambition, but fewer invisible liabilities. Not simplicity achieved by hiding complexity, but structure that makes complexity easier to see. Not software that only its author can repair, but software whose logic and failures remain available to the people responsible for it.

Reliable and efficient. Organized and readable. Hard things fast to realize. Dangerous things difficult to do accidentally. Broad freedom in outcomes, strong opinions about implementation.

And no shortcut that leaves the final product worse.

PART I · CHAPTER 03

One Platform, One Flow Model

Understand how Flow-Like Apps, Flows, Boards, Events, data, permissions, authoring surfaces, and local or remote execution fit into one model.

A language does not make a platform.

FlowScript can describe the logic of a process, but logic alone does not decide where files belong, who may run the process, which version an API should invoke, or where execution should happen. Those concerns live around the language, and they are part of whether an application can be understood and operated.

Before we write our first Flow, we therefore need a map:

```
Flow-Like
└── App
    ├── Flows (each persisted as a Board)
    │   └── Pages and interfaces
    ├── App Events
    ├── Storage and Data Studio
    ├── Packages
    └── Members, roles, and publication settings
```

This is a conceptual map, not a deployment diagram. Flow-Like is the platform. An App is a project inside it. A Flow is the author-facing name for an executable workflow; internally, that same unit is persisted as a Board. Studio and FlowScript are two ways to author the same model.

Once those terms are clear, the rest of the system becomes much easier to place.

3.1 Flow-Like, App, Flow, and Board

Flow-Like is the whole environment in which applications are created, shared, run, and observed. It includes the authoring clients, the node catalog, the Rust execution engine, data and storage services, application interfaces, identity and permission boundaries, and the remote services used by online deployments.

That makes Flow-Like larger than FlowScript. FlowScript is one language inside the platform; it is not the name of the runtime, the visual editor, or the application a user eventually opens.

An **App** is the project and governance boundary. It groups the things that belong to one solution: its Flows, Events, pages, files, structured data, packages, members, roles, and release settings. Our Incident Triage example will be an App. Its classification logic will be a Flow. A quick action or API that invokes that logic will be an App Event.

The word *App* does not imply that the result must be a mobile or desktop interface. An App might contain an automation, an agent, a data pipeline, an API, a form, an analytical tool, or several related experiences. The term tells us where ownership and related resources meet, not what shape the final interface must take.

A **Flow** is an executable process inside an App. This is the term authors normally use when talking about the logic they are building: “run the triage Flow,” “open the ingestion Flow,” or “pin the published Event to version two of this Flow.”

Internally, that Flow is stored as a **Board**. The Board contains the graph: nodes, pins, connections, variables, layers, comments, version information, execution settings, and other metadata needed to edit and run it. *Flow* is the user-facing unit of logic; *Board* is the precise name for its persisted graph representation.

The distinction may feel small at first, but it prevents confusion later. An App can contain many Flows, and a Flow can own multiple Pages. Each Flow is persisted as a Board. A workflow-backed App Event can point to an entry node on a particular version of that Board. The runtime loads the selected Board and executes its graph.

That chain is the spine of the platform:

> *An App owns the solution. A Flow expresses one executable process. A Board persists that process as a graph.*

3.2 Studio and FlowScript

A Board has two serious authoring surfaces.

Flow-Like Studio is the visual surface. It places nodes on a canvas and connects them with typed wires. Studio is particularly good at revealing locality: where a value originates, where execution branches, which operations form one layer, and which node produced a result or failure.

FlowScript is the textual surface. It expresses the same logic with declarations, calls, expressions, branches, loops, functions, and event-entry declarations. Text is denser than a canvas. It is usually faster to search, review, compare, and change when a Flow becomes large.

The relationship between them is easy to describe incorrectly. FlowScript is not a program that runs beside the Board, and Studio is not a diagram generated after the real work is done.

The durable formulation is:

> *Studio and FlowScript are equal authoring surfaces over one underlying Flow model.*

In the current implementation, the Board is what Flow-Like persists. Opening FlowScript renders that Board as editable text. Applying an edit parses the text, reconciles it with the existing graph, and produces Board changes. The Rust runtime then executes the resulting graph. FlowScript does not leave the platform or bypass its execution model.

This gives each visible operation one identity across both surfaces. A node selected in Studio can be found in the source. An anchored statement edited in FlowScript can update the same node rather than creating

an unrelated replacement. Later chapters will examine the anchors, reconciliation plan, and deletion protections that make this possible.

The contract is ambitious because both directions must remain honest. Formatting a graph as text is not enough; applying edited text must preserve the intended graph identity and the Board details that source does not need to spell out. When the current implementation cannot apply a construct safely, it should report the edge rather than silently inventing a different program. Such a mismatch is something to report and improve, not a reason to maintain separate textual and visual versions by hand.

For an author, the practical rule is simpler: choose the surface that makes the current question easiest, then move to the other without changing programs.

3.3 Nodes, pins, wires, and layers

A **node** is a typed operation. It may transform a value, call a service, write a file, branch execution, read a variable, produce an interface message, or mark the entry to a Flow.

Calling a node in FlowScript can resemble calling a function, but a node carries more platform meaning than an arbitrary function. It has a catalog identity, documentation, declared inputs and outputs, execution behavior, and potentially requirements such as OAuth scopes, local-only execution, or WASM capabilities. That metadata lets the authoring tools, runtime, and governance systems reason about the same building block.

A node communicates through **pins**. Input pins accept values or execution. Output pins produce them. Data pins declare a data type and a value shape, such as one value, an array, or a set. Structured pins can additionally carry a schema. Defaults and constraints may narrow what an input accepts.

A **wire** connects two compatible pins. Data wires carry values. Execution wires carry control order. Keeping those two relationships separate is fundamental.

Consider an operation that writes an incident record. The record itself may arrive through a data wire, but the write should happen only after a preceding validation step succeeds. The data connection answers *what value is available?* The execution connection answers *when should this side effect happen?*

This distinction also separates pure and impure work. A **pure node** has no execution pins. It is evaluated when a downstream operation needs its value. An **impure node** participates in the execution path because it performs work whose order matters. A string conversion can often be demand-driven. Sending a message, updating state, or calling an external service usually cannot.

FlowScript compresses many of these relationships into familiar source forms, but it does not erase them. An expression may lower to a pure node. A statement may represent an impure node on an execution chain. A branch or loop becomes visible control structure in both representations. Chapter 5 will follow these mappings in detail.

Finally, **layers** organize a graph in depth. A group of nodes can be collapsed behind a named boundary with typed inputs and outputs. From the outside, the layer behaves like a higher-level operation; inside, its implementation remains a visible graph. Function layers also give FlowScript callable functions. Layers let the top level communicate the shape of a process without hiding the implementation in an arbitrary code block.

3.4 Events and application surfaces

Flow-Like uses the word *Event* at two connected but distinct levels.

An **event node** lives inside a Flow. It is an entry point into the graph. Its output pins define the values available when execution begins, and its kind determines which App Event types can expose it.

An **App Event** lives at the App level. A Page-target Event opens a Page directly. A workflow-backed Event selects a Flow, an event node, a Local or Remote execution location, and either the latest draft or a numbered Flow version. Type-specific settings may then add a route, schedule, authentication, payload, or interface configuration.

Depending on the event node and environment, that path might be a quick action, chat interface, generated form, API endpoint, schedule, deep link, daemon, REST surface, MCP server, or an integration such as email or messaging. Page-target Events can open a visual page directly. Not every event type is available for every node or execution location; the editor constrains the combinations.

This separation is useful because one piece of logic can serve more than one surface. A Generic Event node might be exposed as a form for a person and as an API for another system. Each workflow-backed App Event can have its own applicable configuration and version choice while still entering the same Flow.

It also creates a clean release boundary. An author can continue editing the latest Flow while a production-facing Event remains pinned to an immutable version. “What logic are we building?” and “What entry point is currently live?” become related questions rather than the same setting.

When this book says **event node**, look inside the Flow. When it says **App Event**, look at the configured bridge between that Flow and its caller.

3.5 Apps, people, permissions, and collaboration

The App is also where a private experiment becomes shared software.

An offline App can remain on one device and operate without platform sign-in. This is useful for local automation, personal experiments, device-bound capabilities, and work that should not be uploaded. An online App is stored through a configured Flow-Like backend and can participate in web access, multi-device use, team membership, remote Events, and publication.

Online Apps begin Private. Prototype enables collaboration subject to a deployment-configured member limit. From Prototype, an owner can request either Public Request or Public; both transitions enter publication review. The important idea is not one permanent numerical limit or a rigid ladder. Personal work can begin with little ceremony, while wider distribution creates stronger review and ownership decisions.

Within an App, roles and rights determine what members may see or do. Invitations, default roles, App ownership, Flow access, and publication belong to the project rather than to an unrelated spreadsheet or ticket system. Versions, templates, and permitted forks likewise retain their relationship to the application they describe.

Several security boundaries meet here, and they should not be collapsed into one vague word such as *permission*. Platform identity answers who is signed in. App membership answers what that person may do in the project. Event authentication answers who may invoke an exposed entry point. Node and WASM capabilities answer what an operation may request during a run. Runtime credentials answer which external resource that operation can actually reach.

Keeping those boundaries in one platform makes them easier to relate. It does not make every policy automatic or every published App safe. Governance can derive useful evidence from the program structure, packages, and executions, but people still decide what their organization permits and which evidence is sufficient.

3.6 Data Studio and app-owned state

Useful logic needs something to work on.

Every App can own files that its Flows read and write. Shared App storage holds project assets; user storage provides a private per-user area within the App. Offline and online Apps persist those files through different configured stores, but the logical relationship remains the same: the data belongs to the application that uses it.

Data Studio is the App's workspace for structured data. Its native tables can be created directly or populated by Flows. Authors can inspect rows, schemas, and indexes, run SQL, retain reusable local queries where the current query model permits it, and view results as tables, charts, graphs, or JSON.

On top of those tables, an ontology can describe the domain in its own vocabulary. Object types turn rows into concepts such as `Incident`, `Machine`, or `Customer`. Relationships connect those concepts. Object

views decide what matters when a person inspects one object. Governed actions define the operations available on it.

This matters because a database schema and a business model are not the same explanation. A column may store the right value while saying little about what that value means or which actions are valid. The ontology layer gives domain experts and application surfaces a shared semantic model without requiring the underlying data to be copied into a separate knowledge system.

Keeping data, actions, and Flows inside one App also improves traceability. An App can connect a Flow that ingests a file or updates a table to a governed ontology action and an Event without pretending that these are unrelated projects. The platform can retain more of the context needed to document and govern that path.

The storage backend still matters. Local files, object stores, relational data, and analytical workloads have different operational properties. “App-owned” describes the logical boundary; it is not a claim that every provider has identical latency, scale, retention, or recovery guarantees.

3.7 One runtime contract, several places to run

The same Flow model can participate in more than one execution environment.

On Desktop, a compatible Flow can run locally through the Rust runtime. In an online App, it can run remotely through the API and executor path. A Flow whose execution mode is **Hybrid** may run locally when invoked from Desktop and remotely when invoked through the web or server path. Hybrid does not divide one graph between a laptop and a server; it allows the whole Flow to run in either suitable environment.

A workflow-backed App Event is more specific. It is bound to **Local** or **Remote**, never Hybrid, because a configured endpoint, schedule, or interface needs an unambiguous execution location. A Flow locked to Local or Remote forces its Events to use the matching location; a Hybrid Flow permits either. A remote Event can run while the author’s Desktop is closed.

A local Event can use capabilities attached to that device. Nodes that depend on local paths, installed software, hardware, or other device features can require local execution.

The broad runtime path remains recognizable in either case. Flow-Like loads the selected Board and, where requested, its pinned version. It resolves the node catalog, payload, variables, caller context, credentials, and required stores. The execution context evaluates data dependencies and follows execution pins through the graph. During the run, it can emit logs, progress, state changes, interface messages, results, and stored artifacts.

That is the **runtime contract**: a shared graph and execution model, surrounded by defined ways to obtain data, credentials, context, storage, and evidence. It is what lets an author reason about the same Flow before choosing its final operating environment.

It does not mean every environment is interchangeable. A local device and a remote executor have different capabilities and trust boundaries. Offline Apps and online Apps have different collaboration behavior. Deployment backends can use different queues, object stores, databases, containers, or cloud functions. Provider-specific implementations may have different maturity, scaling characteristics, and operating requirements.

The repository contains local and self-hosted deployment paths, including Docker Compose and Kubernetes, as well as ongoing provider-specific backend implementations. Their presence does not by itself establish equal support or production readiness. Later chapters will classify targets by release and distinguish supported operation from preview or architectural intent. For now, the reliable claim is smaller: Flow-Like aims to keep the application model and execution contract stable while allowing the surrounding infrastructure to vary.

We now have the complete map needed for our first program.

We will create an App named **Incident Triage**. Inside it, we will build one Flow. Its Board will contain typed nodes connected by data and execution wires. We will author that Board in Studio and FlowScript. An event node

will define where execution begins, and an App Event will later decide how someone invokes it. Its data and evidence will remain attached to the App, and the runtime will execute the same graph we can inspect.

The next chapter turns that map into something that runs.

PART I · CHAPTER 04

First Flow: Incident Triage in Two Views

Build a deterministic incident triage Flow, inspect its node graph, edit it in FlowScript and Studio, trace failures, and save a tested version.

It is time to build something.

Our first Flow will not call an API, open a database, or ask an AI model to make a judgment. It will accept one incident report, normalize the text, classify it with a deterministic rule, and record the decision at the appropriate log level.

That narrow scope is intentional. We want to see the relationship between FlowScript and the Board without credentials, network failures, or probabilistic behavior getting in the way. When the Flow behaves differently, we should be able to explain the difference from its input and six visible nodes.

By the end of the chapter, we will have changed the same program from both authoring surfaces, inspected its run evidence, and saved a known-good version.

Release check: The source follows the current renderer's canonical event-parameter and import conventions and round-trips through the parser and renderer. Its six mappings have been checked against generated node declarations and direct tests of the constituent reconciliation behavior. Exact canvas gestures, a combined end-to-end run, screenshots, and the malformed-input trace must be captured against one named Flow-Like release before publication. This draft does not invent release-specific interface behavior.

4.1 The contract: accept, classify, respond

The smallest useful incident record for this exercise has one field:

```
report: string
```

The Generic Event also exposes its built-in `payload: Struct`. That payload is the original event object or envelope. We will leave it unused and work through the named, typed `report` output so the contract stays obvious.

The Flow applies one rule:

1. Remove whitespace from the beginning and end of the report.
2. Look for the phrase `production is on hold`, without regard to letter case.
3. If the phrase occurs, write the normalized report as an Error log.
4. Otherwise, write it as an Info log.

"Respond" means writing evidence to the run log in this first example. The Flow does not yet return a value to an API caller, send a notification, or create an incident record in Data Studio. Those would be reasonable next steps, but each would add concepts that are not needed to understand the dual-view loop.

The rule is deliberately literal. It is not a complete severity model, and it does not pretend to understand language. Given the same string, it produces the same branch decision every time. That makes it a good first program and a useful test fixture.

Three inputs will matter later:

CASE	INPUT
Normal	`Database latency is elevated, but production continues.`
Urgent	`PRODUCTION IS ON HOLD after an interface timeout.`
Malformed	A payload in which `report` is not a string, such as `{"report": 42}`

We can already predict the domain behavior of the first two cases. The normal report does not contain the phrase and follows the Info path. The urgent report loses its surrounding spaces, matches despite its capitalization, and follows the Error path.

The malformed case is different. Its correct boundary depends on the release and invocation surface. A typed form may prevent the value from being submitted. An API adapter may reject the payload before the Board starts. If the value reaches the graph, the string operation may reject it there. We will observe that boundary rather than writing the desired outcome as if it had already happened.

One more distinction matters: the `log::error` operation records an Error-level message. Its node completes successfully; it does not throw an exception by itself. Some run lists may still classify or color the run from its highest log severity. The urgent path is a successful classification whose result happens to be serious. A true node failure is a different event, and its evidence should be read differently.

4.2 Build it visually

Create an App named **Incident Triage**, then create one Flow inside it. The initial Board contains six operations:

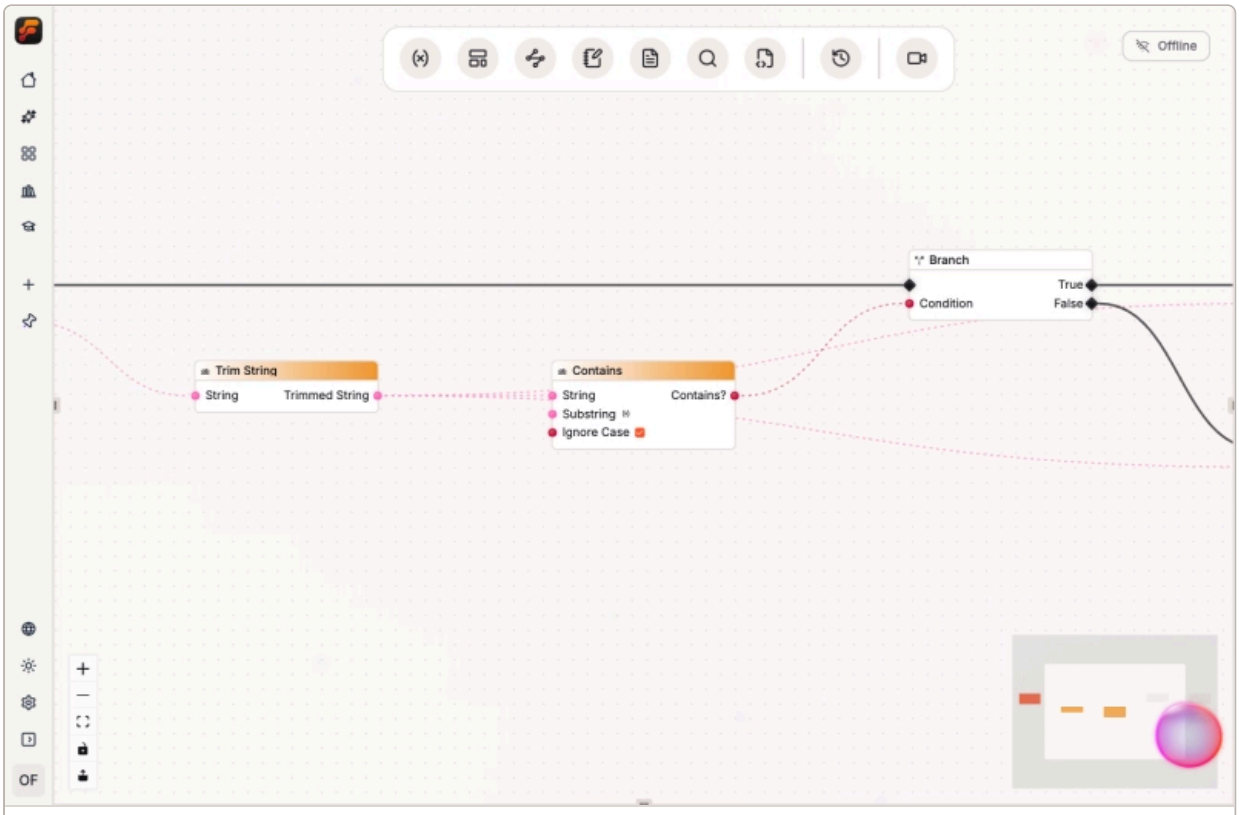
NODE	CATALOG IDENTITY	RESPONSIBILITY
Generic Event	`events_generic`	Starts the Flow and exposes `report: string`
Trim String	`string_trim`	Removes leading and trailing whitespace
Contains	`string_contains`	Checks the normalized report for the incident phrase
Branch	`control_branch`	Selects the urgent or normal execution path
Log Error	`log_error`	Records an urgent report
Print Info	`log_info`	Records a normal report

The displayed names can evolve. The catalog identities are the version-matched implementation references used to verify the source mappings.

Begin with the Generic Event. Rename it **Triage Incident**, which gives the entry its `triageIncident` source alias, and add a string output named `report`. A Generic Event already provides its execution output and `payload`; FlowScript can additionally define named, typed outputs such as this one.

Connect the `report` value to the String input of Trim String. Connect Trimmed String to the String input of Contains. Configure Contains with the substring `production is on hold`, and enable its case-insensitive comparison.

The data side of the graph is now visible in Studio:

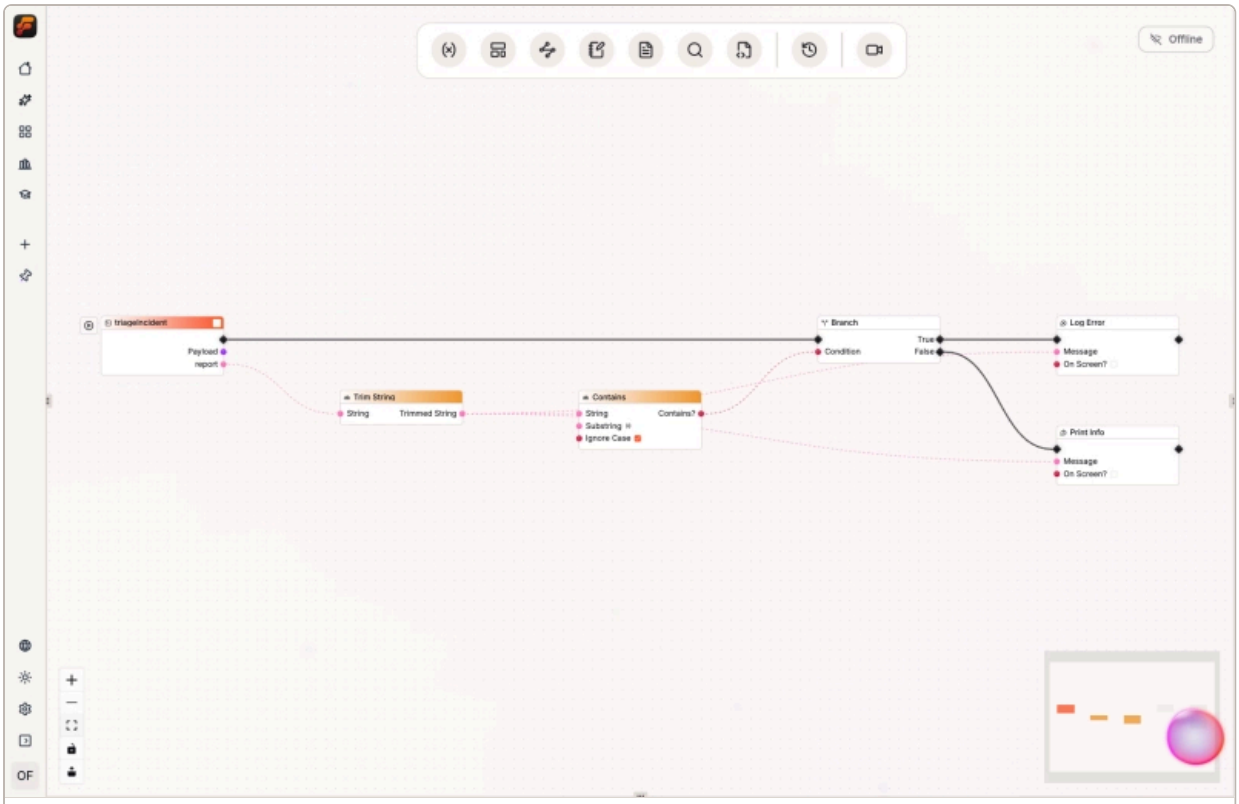


A focused crop of the first data path after FlowScript reconciliation and balanced auto-layout. Open the image to inspect the pins at full size.

The execution side is shorter. Connect the Generic Event execution output to the Branch input. Connect the Branch's True output to Log Error and its False output to the Info log node.

Finally, connect the normalized string from Trim String to the Message input of both log nodes. Set the on-screen notification option to false for each. We want recorded run evidence, not a temporary toast.

The completed graph therefore has two kinds of wires:



The complete six-node FlowScript rendered in Studio. Solid dark wires carry execution; colored dotted wires carry typed data.

Notice what is absent: Trim String and Contains do not need execution wires. They are pure operations. When Branch needs its condition, the runtime evaluates the data dependencies that produce it. The Branch and log nodes are impure because their position in the execution path matters.

This separation answers two different questions. Data wires answer, “Where does this value come from?” Execution wires answer, “When does this work happen?” A graph that mixes those questions into one kind of connection becomes difficult to reason about as soon as side effects appear.

At this point, pause before opening the text view. Read the graph from left to right and say the rule aloud:

> *When an incident arrives, trim its report. If the normalized report contains “production is on hold,” log an error. Otherwise, log information.*

If the visible graph does not communicate that sentence, improve its placement before moving on. Correct wiring is necessary; readable organization is part of the result.

4.3 Read the same Flow as source

Open the FlowScript view of the Board. Ignoring the identity anchors that the editor may append, the program is:

```
use log::*

eventsGeneric triageIncident(payload: Struct, report: string) {
    const normalized = report.trim()
    if (normalized.contains({ substring: "production is on hold", ignoreCase: true })) {
        error({ message: normalized, toast: false })
    } else {
        info({ message: normalized, toast: false })
    }
}
```

This is not a handwritten reimplementation of the graph. It is the textual representation of that Board.

The import at the top tells FlowScript that unqualified logging calls come from the `log` namespace:

```
use log::*
```

The renderer derives this glob import because the Flow uses multiple static calls from the same namespace. Fully qualified calls such as `log::error(...)` remain valid source, but the Board's canonical rendering uses the import and the shorter aliases here.

The event declaration describes the entry node:

```
eventsGeneric triageIncident(payload: Struct, report: string) {
```

`eventsGeneric` identifies the Generic Event kind. `triageIncident` is the name we give this entry. `payload: Struct` is the event node's built-in payload output. The additional parameter `report: string` is the typed output we defined for this Flow.

The next line contains one binding and one method-shaped node call:

```
const normalized = report.trim()
```

`report.trim()` represents the Trim String node. FlowScript lets nodes with a receiver pin use method syntax, so the value on the left of the dot supplies the node's String input. The default output becomes `normalized`.

This does not create a hidden JavaScript string operation. It still names a catalog node, and that node remains visible on the Board.

The condition contains two more operations:

```
if (normalized.contains({ substring: "production is on hold", ignoreCase: true })) {
```

`normalized.contains(...)` is the Contains node. The receiver supplies its String input. The object supplies the remaining pins: the substring and the case-comparison option. Its boolean output supplies the Branch condition.

The `if` block is the Branch node rendered as familiar control flow. The first block corresponds to its True execution output; the `else` block corresponds to False.

Finally, each statement inside those blocks is an impure logging node:

```
error({ message: normalized, toast: false })
info({ message: normalized, toast: false })
```

The preceding `use log::*` makes these aliases available without a prefix. In the fully qualified spelling, `::` separates the `log` namespace from the node alias. A dot means field or method access on a value; `::` addresses an operation through its namespace. The object keys are the node's input pins.

Count the graph behind the source: one event, one trim, one contains check, one branch, and two logs. Six nodes.

The editable view may include trailing comments such as `//@n:...`. Those are identity anchors. They let reconciliation associate a statement with the node already on the Board. The book omits them for readability, but an author should preserve them while editing unless a deletion is intentional.

FlowScript accepts semicolons but does not require them. Canonical rendering omits them. That formatting choice does not change the Board.

4.4 Change text, watch the graph

Change the urgent phrase in the Contains call:

```
substring: "customer orders are blocked"
```

Do not apply the edit immediately. First inspect the pending reconciliation preview. The editor checks the source against the current Board and produces a command plan without mutating the persisted graph.

For this change, the important question is not the exact number of internal commands. That can vary as reconciliation and formatting evolve. The important shape is that an existing node is updated. The preview should not propose deleting the Flow or replacing all six nodes. If it does, stop and inspect the source and its anchors.

Apply the edit to the Board. Then locate the Contains node and inspect its configured substring. The graph should now express the new rule while its structure and node identities remain intact.

This is a small edit, but it demonstrates the essential contract. You changed code, yet the result was not a separate textual program. FlowScript was parsed and reconciled into a Board change.

Use the editor's source-to-node navigation affordance where it is available in your release. The release verification pass for this chapter should capture both the pending command plan and the changed node on the canvas.

Once you have seen the round trip, restore the original phrase and apply it again. The remaining examples use `production is on hold`.

4.5 Change the graph, watch the text

Now travel in the other direction.

On the urgent execution path, insert a Log Warning node before Log Error. Give it the normalized report as its message and keep its on-screen notification disabled. The True path should enter Log Warning and then continue to Log Error.

After the Board change has been saved, re-render FlowScript from the Board. The urgent branch should be semantically equivalent to:

```
if (normalized.contains({ substring: "production is on hold", ignoreCase: true })) {
    warn({ message: normalized, toast: false })
    error({ message: normalized, toast: false })
} else {
    info({ message: normalized, toast: false })
}
```

The final rendered text may include anchors and release-specific canonical formatting. Verify the actual output rather than copying those details from this draft.

This direction of the loop is just as important as the first. The visual edit did not create an annotation beside the source. It changed the Board, and the source now describes that change.

Avoid editing stale text while changing the canvas. The current editor guards against applying a draft when the Board has changed behind it; nevertheless, the simplest working habit is to finish or reset a text edit before switching surfaces, then re-render from the Board after a visual change.

For the chapter fixture, remove the temporary warning again and confirm that the source has returned to the six-node baseline. The warning was useful for proving the reverse trip. It is not part of our classification contract.

4.6 Break it on purpose

Run the Flow first with the normal report:

```
{
  "report": "Database latency is elevated, but production continues."
}
```

Select the resulting run and inspect its logs. The rule predicts an Info message containing the normalized report. The Error log node should not execute.

Next, run the urgent case:

```
{
  "report": "  PRODUCTION IS ON HOLD after an interface timeout.  "
}
```

The rule predicts that Trim String removes the surrounding spaces, Contains returns true despite the capitalization, and the Branch follows its True output. The resulting message should be recorded at Error level without an on-screen toast.

Again, this is not a thrown failure. The Log Error node completed the action we asked it to perform. Some run lists classify or color a run from its highest log level, while remote execution keeps completion status and log severity as separate fields. An intentional Error log can therefore look similar to a failed run at a glance. Read the node-attributed evidence before deciding what happened.

Now try the malformed payload:

```
{
  "report": 42
}
```

This is where the chapter must follow evidence instead of aspiration.

Before final publication, run this case through the same invocation surface used in the screenshots and record the actual boundary:

- If the interface refuses the payload, document the typed input rejection.
- If an App Event adapter rejects it, document that the Board did not start.

- If it reaches the Board and a string operation fails, record the failing node and its error.
- If the value is coerced or defaulted, document that behavior and decide whether the fixture needs a different deliberately failing input.

Do not claim that Trim String failed merely because it is the first string node. A platform boundary that rejects the wrong type earlier may be the safer and more accurate result.

When a true node failure is available, open its run, find the relevant log entry, and use the node-navigation action to return to the responsible block. Capture the input, Flow version, run identifier, log level, message, and focused node. Those details turn “it failed somewhere” into reproducible evidence.

This is the flight-recorder role of the run log. A useful trace preserves more than a sentence saying that execution stopped. It relates the evidence to a run, a version, and a node identity. An unfamiliar responder can start at the operation involved and trace its incoming values and surrounding execution path.

Our six-node Flow is too small to make that navigation feel necessary. That is precisely why it is worth learning here. In a Board with hundreds of nodes, beginning at the responsible operation rather than at the application boundary can save the first hour of an investigation.

4.7 Save the first known-good version

Return the Board to the six-node baseline and rerun the verified test matrix for the release you are using. At minimum, preserve the normal and urgent runs. Add the malformed-input case once its expected boundary has been established and documented.

Then create an immutable Flow version. Use the semantic increment that matches your release policy; for a compatible correction to an existing Flow, that would normally be a Patch version. The current draft remains editable, while the numbered version becomes a read-only snapshot.

Record four things with the snapshot:

- the canonical FlowScript rendered from the Board;
- the test inputs and their expected log levels;
- the Flow-Like release against which they were run; and
- the observed malformed-input boundary.

A production-facing App Event can later target that numbered version instead of Latest. New work can continue on the draft without silently changing the entry point used by callers. When the next version is ready, test it and deliberately move the Event.

We have now completed the full loop.

We built a graph and read it as source. We changed the source and updated an existing node. We changed the graph and recovered the corresponding source. We ran deterministic cases, kept application-level severity distinct from runtime failure, and preserved a tested snapshot.

Most importantly, there was never a visual workflow and a separate codebase to keep in sync.

There was one Flow, seen from two sides.

PART II

Thinking and Writing in Flows

Learn the language through the graph it represents: values, operations, control flow, state, and governed runtime boundaries.

PART II · CHAPTER 05

Nodes, Pins, Wires, and Execution

Learn how Flow-Like separates typed data flow from execution order, evaluates pure nodes on demand, and represents sequence, parallelism, and layers.

The six nodes in our Incident Triage Flow told two stories at once.

One story carried the report through Trim String and Contains into the Branch condition. The other started at the event, entered Branch, and continued through exactly one logging path. The first story answered, “Which values does this decision need?” The second answered, “Which consequential operation runs next?”

That separation is the foundation of Flow-Like’s execution model. It is also why a Flow can be compact in FlowScript without becoming opaque on the Board. An expression can describe a chain of value-producing nodes. Statement order and control blocks can describe execution wiring. The two views compress the graph differently, but they do not disagree about what runs.

This chapter gives us the vocabulary to reason about that contract before the language becomes more complex. When a Flow surprises us, we will read it twice: forward along execution and backward through data.

Release check: The static contracts in this chapter are verified against the current Board, node, pin, FlowScript reconciliation, and Rust executor implementations. Before publication, capture the exact pin shapes, wire styles, **Handle Errors** gesture, Sequence behavior, and Parallel Execution behavior in one named Flow-Like release. In particular, do not turn a pin’s configured default into a claim that every value observed during a run is persisted: runtime pin values are deliberately transient.

5.1 A node is a typed operation

A node is the smallest operation that Flow-Like's authoring tools, runtime, and operators can all name.

For an author, a node might trim a string, send a request, write a record, call a model, or choose an execution path. In FlowScript, using one often looks like a normal function or method call:

```
const normalized = report.trim()
```

That familiar spelling is useful, but `trim()` has not become an invisible language primitive. It still resolves to a catalog node. Opening the same Flow in Studio reveals that operation, its pins, and its place in the graph.

The catalog identity matters because a node carries a contract beyond "some code runs here." A node can declare:

- a machine-readable type name and a human-readable name, description, category, icon, and documentation;
- typed input and output pins, including defaults, schemas, and constraints;
- whether it takes part in execution or only produces values;
- schema/migration version and environment restrictions;
- OAuth requirements and, for external WASM nodes, requested capabilities; and
- optional privacy, security, performance, governance, reliability, and cost signals.

Not every node supplies every item, and metadata is not proof that an implementation is correct. It is nevertheless a substantial difference from an arbitrary block of inline code. The platform has a declared surface it can inspect before execution and an identity it can connect to run evidence afterward.

This is where constraints create useful leverage. A custom WASM node declares specific requested capabilities such as outbound networking, storage access, model access, or function calls instead of presenting itself as

an undifferentiated package. A built-in or packaged node can also expose quality signals that contribute to reviewing the App as a whole. Later chapters will examine which boundaries enforce those declarations in each execution mode. For now, the important point is that the operation remains visible to the platform.

That visibility gives three people a shared unit of conversation:

- A domain expert can say, "This is the operation that sends the case to finance."
- A developer can inspect the node's input, output, implementation, and failure contract.
- An operator can begin with node-attributed log evidence instead of guessing which subsystem was active.

The node is therefore more than a box and more than a function call. It is a typed, inspectable building block whose visual identity, source representation, and runtime identity meet.

There is one terminology trap to avoid. *Standard* does not mean "built in," and *pure* does not mean "small." In the current runtime, a node is structurally pure when it has no Execution pins. A node with an Execution pin is impure: its position in control flow matters. Either kind may come from the standard catalog or an approved package.

5.2 Data pins carry values

Pins are a node's public boundary. An input pin states what the operation accepts. An output pin states what it produces. A data wire connects an output to a compatible input and answers one question:

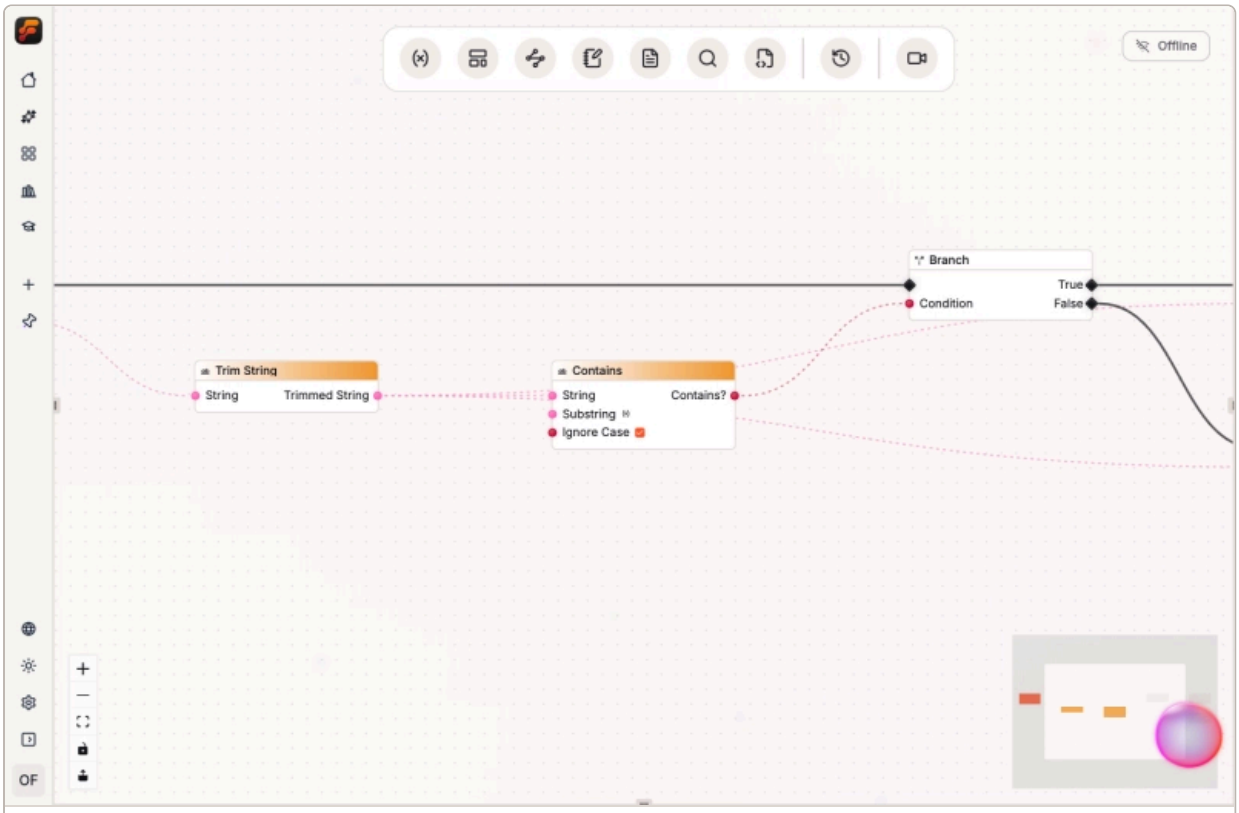
> *Where will this input get its value?*

Direction is part of the contract. Two pins both labeled `Message` are not interchangeable if one accepts a value and the other produces one. Studio rejects input-to-input, output-to-output, and self-connections instead of

asking the runtime to improvise.

Type is the next part. The current graph model distinguishes strings, integers, floats, booleans, dates, paths, bytes, structured values, generic values, and Execution. A separate value shape distinguishes a single value from an array, map, or set. Structured pins can carry a schema so that "object" does not have to mean "anything whatsoever."

Our first Flow already used this system:



Studio exposes direction, type, and fan-out at the pins. Pink dotted wires carry strings; the darker Boolean output supplies Branch.Condition.

The connection from Contains to Branch is valid because the producer and consumer agree on a single Boolean. Connecting the same output directly to a String input should fail during authoring. The rejected wire is not inconvenience added by the editor; it is a data-shape bug found before a production run.

Generic pins provide controlled flexibility. A generic formatter may accept several concrete types. Once a concrete wire resolves that generic, connected pins must remain compatible with the resolved contract. This is type inference, not a suspension of type checking.

Pins can also carry options. A node author may define valid choices, a numeric range or step, a sensitive literal, or rules about enforcing schema and generic value shape. These options help Studio build a better editor for the operation, but the pin is still the underlying contract.

An input can receive its value in two ways:

1. A data wire supplies it from an upstream output.
2. With no upstream source, the node uses a configured default when its contract provides one.

This explains why execution can reach a node whose data wire is absent. Data does not schedule the node, and a missing connection does not automatically mean "do not run." The node may receive its default; if the required value is absent or invalid, evaluation may fail at that node instead.

It is equally important to distinguish configuration from runtime state. The Board persists pin definitions, connections, and configured defaults. During one run, the executor attaches the values it evaluates to an in-memory pin representation. Those runtime values are explicitly excluded from pin serialization. A node-authored log may expose selected evidence, but we must not claim that every value flowing through every pin becomes permanent history.

That is a deliberate boundary. Persisting all values indiscriminately would be expensive and could copy sensitive business data into logs. What evidence a node emits is part of its contract, and what an organization permits in that evidence is part of its policy. We will develop that boundary in the observability chapters.

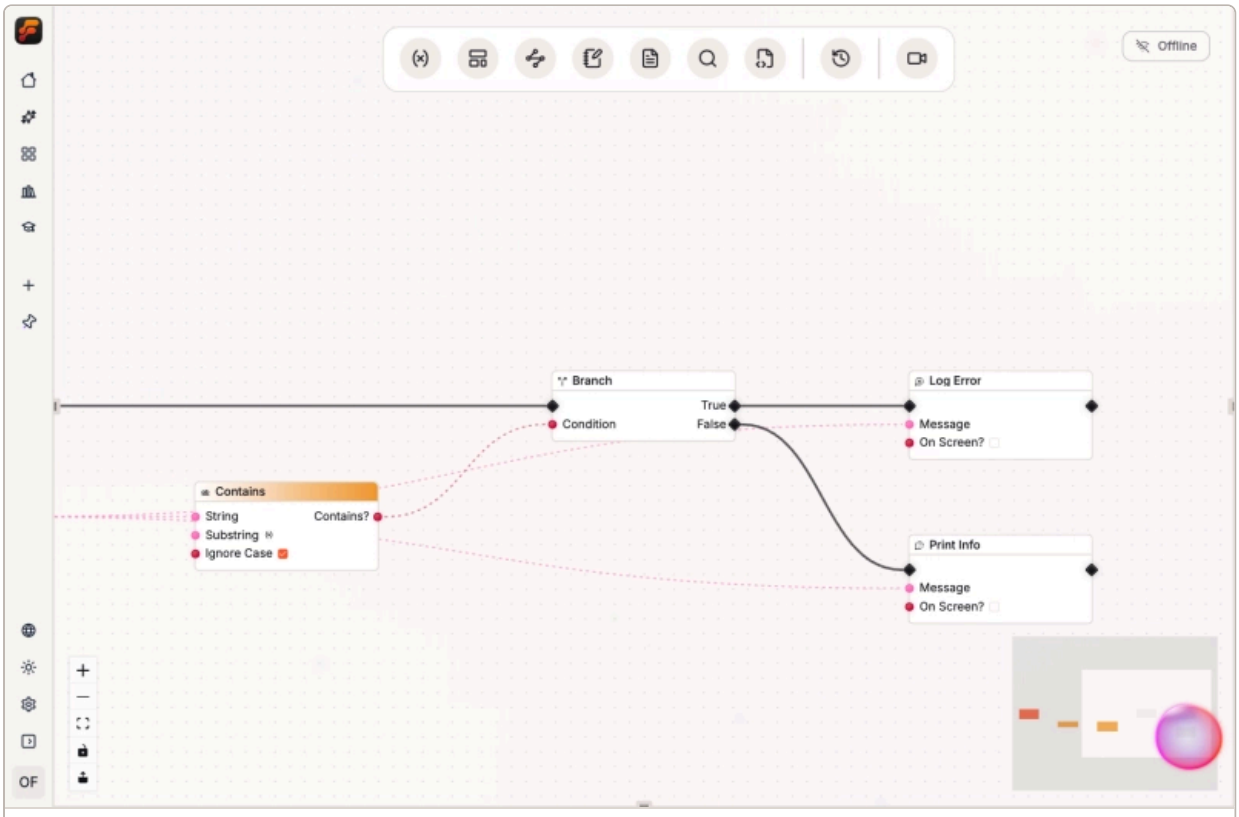
For debugging, the practical rule is to work backward. Begin at a surprising input and follow its data wire to the producer. Repeat until the first surprising value or missing source appears. Looking only at the execution path cannot answer where the value came from.

5.3 Execution pins carry order

Execution pins do not carry application data. They carry permission to proceed.

An execution input answers, “What must activate before this node runs?” An execution output answers, “Which path becomes eligible after this node completes or chooses an outcome?” In the current Studio conventions, these are the diamond pins joined by the execution path. Their exact color and styling are presentation; their Execution type is the durable rule.

Return to the Incident Triage graph:



The solid execution path reaches exactly one logging node; the dotted data wires only supply values.

The event starts control. Branch evaluates its Boolean data input and activates one of two execution outputs. The selected logging node then runs because control reached it—not because its Message input happened to contain a string.

FlowScript renders this graph as familiar control flow:

```
if (normalized.contains({ substring: "production is on hold", ignoreCase: true })) {
    error({ message: normalized, toast: false })
} else {
    info({ message: normalized, toast: false })
}
```

The `if` is not a comment laid over the graph. It represents the Branch node and its labeled True and False execution outputs. The statements inside each block are the impure nodes wired to those outputs.

Ordinary consecutive impure statements similarly represent an execution chain. Their order in source is meaningful because their side effects are meaningful. Sending a message before creating its recipient, charging a customer before validating an order, or writing a completion record before the work finishes are different programs even if every data value is available.

A failed impure node normally stops its own forward path: its ordinary successors are not queued, and the run records that a node failed. Flow-Like also makes recoverable failure visible in the graph.

Some nodes define outcome paths as part of their own contract. The current API Call node, for example, exposes Success and Error execution outputs. A completed HTTP exchange selects Success only for a successful status and selects Error for other responses, while keeping the response available as data. A transport-level failure is a different case: it is a node error rather than a completed response outcome.

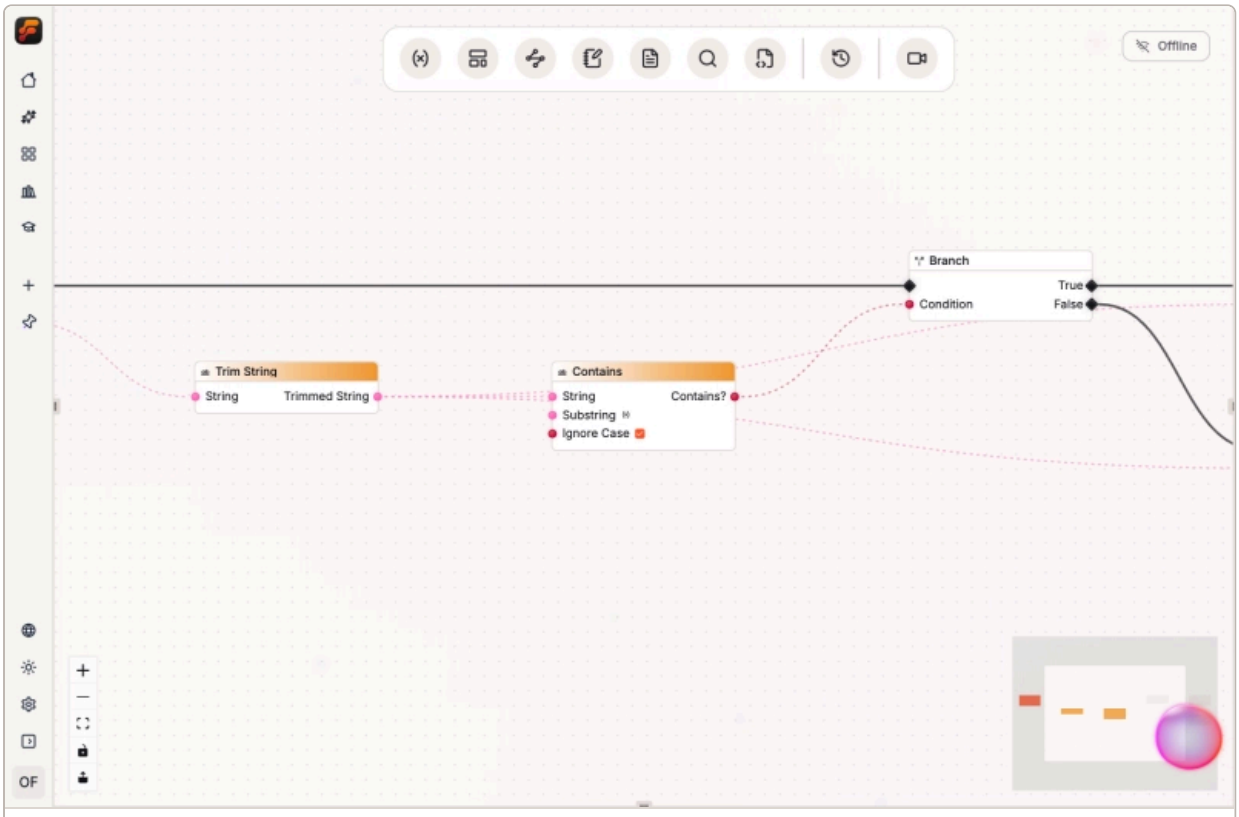
For an unexpected error on any executable node—including one that already models normal Error outcomes—the current Studio toolbar can enable **Handle Errors**. This adds an **On Error** execution output and an Error string output. When a node fails and an On Error path is connected, the runtime executes that handler chain. The original node remains in Error state during the run, and its node-attributed Error evidence remains available afterward; successful handling does not rewrite history and pretend the operation succeeded. It means the Board dealt with the unexpected error rather than allowing it to unwind the run.

That makes remediation part of the program. An integration expected to fail occasionally can route its failure into an operation that creates a ticket, records a fallback, or asks a person to intervene. Whether to recover, retry, escalate, or stop is the Flow author's decision under the organization's policy—not an invisible global guess.

5.4 Pure work is demand-driven

A pure node has data pins and no Execution pins. It does not occupy a place on the control path. Instead, a consumer pulls its output when that output is needed.

In our first Flow, Branch needs its Condition. To obtain it, the executor follows the data dependency to Contains. Contains in turn needs the normalized report, so the executor follows the next dependency to Trim String. The canvas records that backward chain of demand:



When Branch asks for its condition, the runtime pulls the visible pure-data dependencies backward from Contains to Trim String.

The result then travels forward to the consumer. No execution wires are required between those pure operations.

This has several consequences.

First, an unconsumed pure chain does not run merely because an event started. A carefully wired formatter whose output connects to nothing is dead work, not a background task.

Second, a data producer becoming available does not “fire” its consumers. Values are pulled when an executing node asks for an input. This is why a correct data graph cannot rescue an impure node whose execution input

is disconnected.

Third, “pure” is an execution contract, not a promise that the runtime will call the node exactly once. Demand can be zero, one, or multiple evaluations depending on consumers and execution contexts. Work that must happen once, work that costs money, and work with an externally visible effect belongs on an explicit execution path.

The current implementation detects purity structurally: if a node has any pin whose data type is Execution, it is impure; otherwise it is pure. That means the node author has a responsibility to make the declaration honest. Hiding a network call or a database write behind a node with only data pins would make cost and failure appear to be an innocent calculation. Flow-Like can enforce declared WASM capabilities, but pin shape alone cannot prove mathematical referential transparency.

A useful design test is:

> *Would it be harmless if this operation were evaluated zero times or more than once?*

Trimming, comparing, selecting a field, and formatting a value usually pass. Sending, charging, mutating, calling a paid model, or asking an unreliable external system usually fail. Put the latter group on the execution path, where order and failure remain visible.

When debugging, this produces the two-pass method introduced at the start of the chapter:

1. Follow execution forward from the event and ask whether control could reach the node.
2. Follow data backward from the surprising input and ask whether its value could be produced.

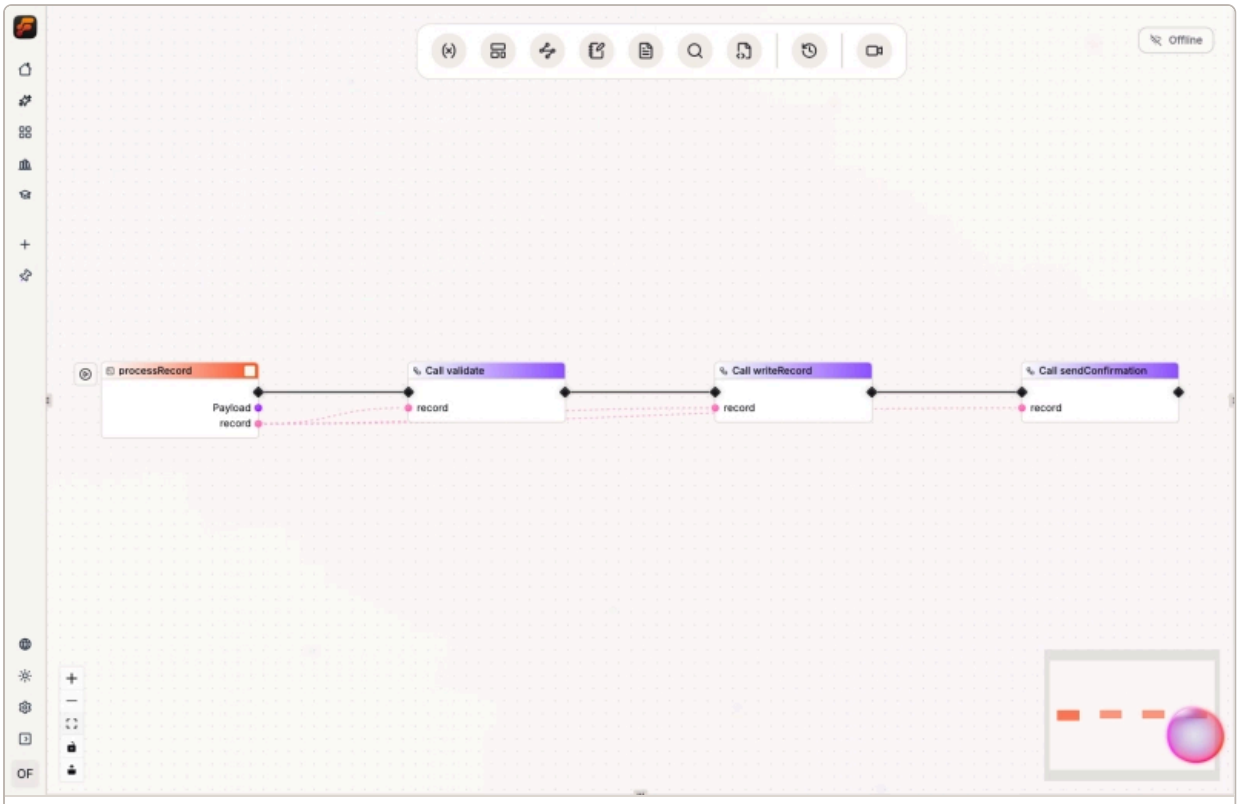
Do not change wires until you know which story is broken.

5.5 Explicit sequence and explicit parallelism

Canvas position is for readers. It has no scheduling authority.

Three nodes arranged left to right do not run in that order because they look like a sentence. Two nodes arranged one above the other do not run concurrently because they look like branches. Execution pins, and the control nodes that activate them, define the behavior.

For ordinary sequential work, draw one unbroken execution chain:



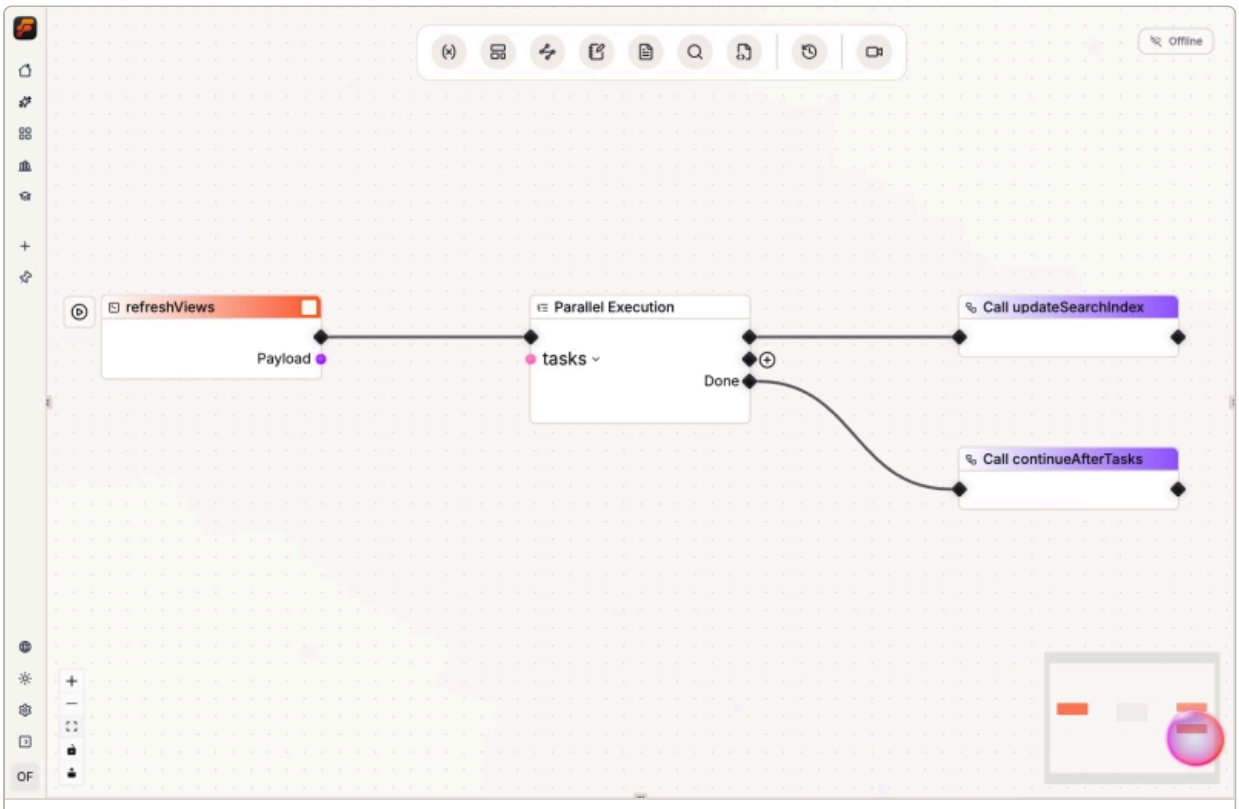
Consecutive impure FlowScript calls reconcile into one explicit execution chain.

In FlowScript, consecutive impure statements express the same intent. Each operation receives control only after the preceding operation's selected continuation is reached.

When several complete branches must run one after another, Flow-Like also has a Sequence control node with ordered outputs. That is different from merely drawing several wires near each other: the node owns the ordering contract.

For concurrent work, use an operation whose contract says *parallel*. The current Parallel Execution node exposes repeatable task outputs and a Done continuation. It starts the connected branches as bounded asynchronous tasks by default, waits for them, and activates Done afterward. Its optional thread mode uses a multi-thread runtime when available and otherwise logs a warning and falls back to task mode. Parallel For Each provides the loop equivalent, including a maximum concurrency input; FlowScript can render its supported form with `@parallel`.

The shape is explicit. The task paths do not have to wire back into Done; the Parallel operation activates that continuation after the task branches settle. The source-authored example below wires one task arm and Done, while Studio can add more task branches through the repeatable output control:



The task tail remains separate from Done. The plus control adds repeatable task outputs on the Board; Done proceeds only after all connected task branches settle.

This is a concurrency promise, not an ordering promise. The branches may finish in any order. They must not race on shared mutable state unless the design makes that safe, and external effects must tolerate the ordering that the Flow declares.

Failure is likewise a control-flow decision. The intended default for visibly parallel branches is isolation between siblings: one failing branch does not prevent the other branches from continuing. The ordinary executor follows that model when several topology targets are ready: it schedules them concurrently, bounded by executor capacity. An unhandled failure ends that branch's successors while the other ready targets continue.

After those siblings settle, the intended aggregate contract is simple: any unhandled child failure makes the whole run's terminal status **Failed**. A failure that reaches and completes an explicit handler remains visible on its node and in attributed logs, but it does not fail the run.

The dedicated Parallel Execution node also collects its child branches and proceeds to Done after they settle. There is a current implementation caveat, however: Parallel Execution and Sequence can record a child error without always propagating that error into the enclosing run's final status. Read the child nodes' attributed logs; do not treat a successful outer control node as proof that every branch succeeded. This aggregation behavior must be tested and documented for the release used by the book.

Do not generalize that into "errors never stop anything." A normal sequential path has different semantics, an unhandled failure has no ordinary continuation, and a purpose-built node may define its own outcomes. If completion of all work is required, put the join in the graph. If one failure must cancel or compensate for the rest, model that policy explicitly and verify the selected nodes' contracts in the target release.

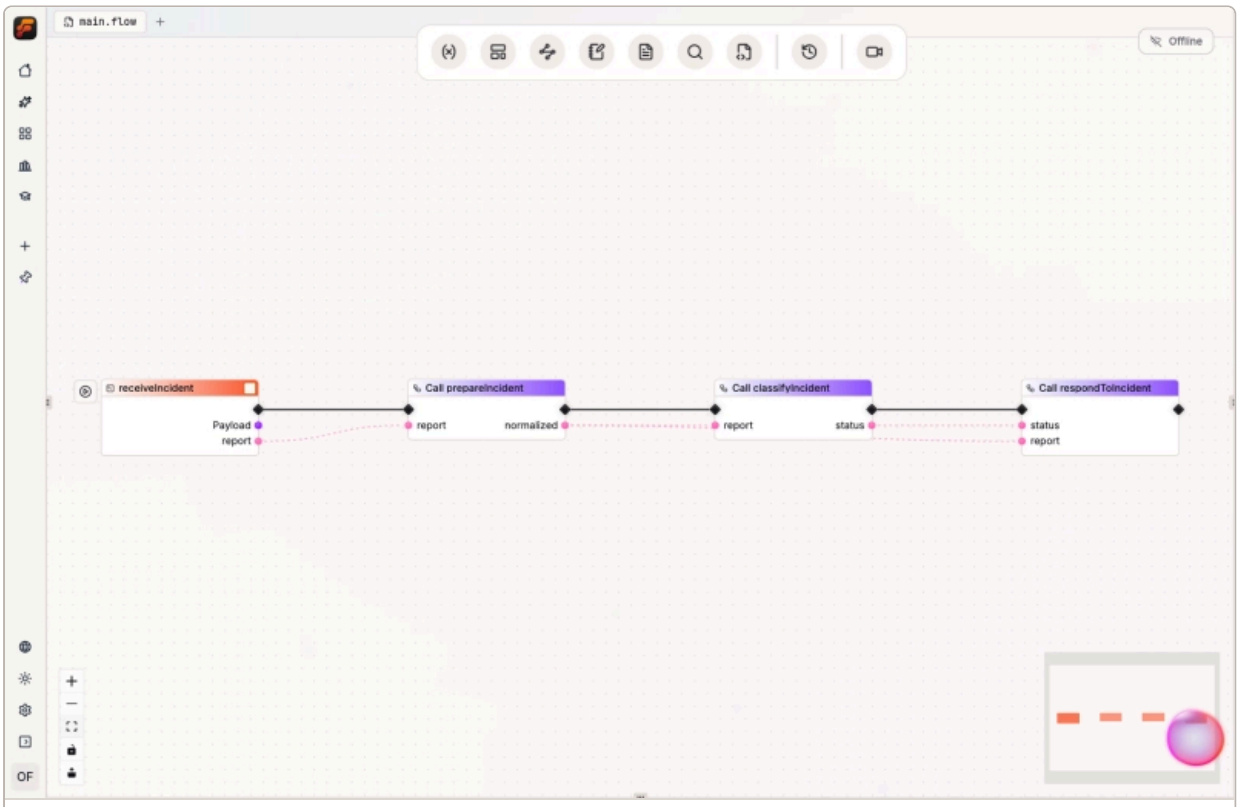
This explicitness costs a few pins and rewards us with an answer during an incident. We can point at the graph and say which work was ordered, which work was concurrent, where the join occurred, and which failure route was selected. Geometry alone could tell us none of that.

5.6 Layers organize; functions are callable

Large Flows need depth, but depth should not create a second hidden program.

A layer folds a group of nodes behind a named, typed boundary. Wires that cross the selection become boundary pins. Open the layer and the same nodes, data dependencies, execution paths, comments, and nested organization remain available. Collapsing changes how much of the graph we see at once; it does not change the rules that run inside it.

Suppose Incident Triage eventually grows from six nodes to sixty. Normalization might include language detection, identifier extraction, enrichment, and validation. We could collapse that inline region into a layer named **Prepare Incident**. Before doing so, compare the related function form: FlowScript can directly author the same top-level sentence as named callable boundaries.



This is deliberately the function form: three Call nodes carry both the execution chain and typed data across named boundaries.

Each purple Call node in that render refers to a function layer, not an ordinary collapsed layer. Opening `prepareIncident` reveals its maintained callable implementation and signature. If the normalization nodes instead remain one inline region, collapsing them into an ordinary **Prepare Incident** layer can present a similar high-level label without making the

region reusable. Its Start and Return boundaries show exactly which typed values and execution paths cross the layer edge. An inner failure remains an inner node failure; the layer is a door, not a black box.

A function is related, but not synonymous. A plain layer organizes one inline region of a Board. A function layer defines logic that callers can invoke from more than one place. Its boundary pins become a callable signature, and a Call Function node represents each call on the graph. FlowScript can therefore give reusable graph logic familiar function syntax without replacing it with hidden source-only code.

The distinction produces a simple rule:

- Use a layer when one part of the Flow needs a clearer name and a lower level of detail.
- Use a function when several callers need the same behavior and one maintained contract.

Callable boundaries carry obligations. Pin names must be unique because callers address the signature. An impure function needs a single execution entry and a valid continuation so the runtime knows where the call begins and how the caller resumes. A genuinely pure function instead returns demanded values without pretending to have an execution path.

Neither mechanism excuses vague architecture. A two-node layer named `Layer 2` hides more than it explains. A function extracted before reuse exists adds an interface without reducing duplication. Name both after their responsibility, keep their boundaries narrow, and treat changes to those boundaries as changes to an API.

We now have the mental model for the rest of FlowScript. Calls resolve to typed nodes. Arguments and results resolve to data pins. Statements and control blocks resolve to execution paths. Pure expressions are evaluated on demand. Sequence and concurrency are declared rather than inferred from layout. Layers manage visual depth; functions add reuse.

The source can become concise because the graph stays precise.

PART II · CHAPTER 06

Anatomy of a FlowScript Document

Read a canonical Flow-Like FlowScript file from use declarations and interfaces through variables, functions, events, identity anchors, and diagnostics.

A FlowScript document is not arranged like a transcript of everything an author happened to do. It is a stable reading of one Board.

Imports establish names. Interfaces describe structured values. Top-level variables describe the Board's per-run and configurable state. Functions define callable layers. Events provide the entries that can start execution. A `detached` block holds an execution chain the Board contains but no entry reaches, one block per chain. Inside those entries and functions, statements describe nodes and control flow.

The order is deliberate:

```
use declarations
interfaces
top-level variables
functions
event entries
detached blocks
```

A `detached` block is what lowering writes when a chain has no reachable entry; nothing inside one runs, so reading it is a finding about the Board rather than a construct to author.

By the end of this chapter, we will be able to look at a complete document and know what kind of graph entity each section represents. We will also know which familiar-looking words have Flow-specific meanings.

Release check: The complete Incident source is a parser-tested canonical fixture at `examples/document-anatomy/anatomy.flow`. A catalog-aware Board reconciliation, an executed run, and the editor's diagnostic presentation must still be captured against the release used for publication. Prose comments are a known parity edge: free-standing `//` comments inside bodies survive parse-and-format, but the current Board reconciler does not persist them as canvas comments, and top-level comments have no durable AST slot.

6.1 Canonical source, not stylistic trivia

FlowScript looks familiar on purpose. Braces, declarations, function calls, object-shaped arguments, interfaces, and control blocks give it a TypeScript-familiar surface. Rust-style `use` declarations and `::` paths solve namespace imports. Decorator-shaped annotations add Flow-specific metadata.

Calling it "mostly TypeScript with a little Rust" is tempting and incomplete. Familiar syntax helps us read; the Board model supplies the semantics. The language is not TypeScript executed by a hidden JavaScript engine, and its decorators are not arbitrary functions evaluated at load time.

The parser accepts more than one spelling for some constructs. The renderer emits one canonical spelling:

- statements may contain semicolons, but canonical output omits them;
- interface properties may omit separators or use commas, but canonical output ends each field with a semicolon;
- strings may be single- or double-quoted, but canonical output uses escaped double quotes;
- indentation defaults to four spaces, and document sections are separated by one blank line;
- comma-separated `use` trees are rendered as one declaration per line; and
- a redundant scalar type annotation is omitted when a literal default infers the same type.

For example, this is accepted input:

```
const urgentPhrase: string = 'production is on hold';

eventsGeneric triageIncident(payload: Struct, report: string) {
  info({ message: report, toast: false });
};
```

Its canonical text uses double quotes, no statement terminators, and four-space indentation:

```
const urgentPhrase = "production is on hold"

eventsGeneric triageIncident(payload: Struct, report: string) {
    info({ message: report, toast: false })
}
```

Canonicalization is not a beauty contest. If the same Board always renders to the same broad shape, then diffs contain more meaning, generated edits have fewer equivalent spellings to choose among, and parse–render–parse can converge instead of churning whitespace.

It also explains why section order may change after formatting. The parser collects top-level declarations by role, and the renderer writes them in the fixed order shown above even if an event appeared before a function in the input. Within Board lowering, variables are sorted by name and stable identity, while function layers and event entries use deterministic identity and entry ordering. Do not encode runtime order by moving top-level declarations around; execution order lives inside their bodies.

The formatter is deliberately narrower than the reconciler. Formatting parses text and renders its syntax without consulting a node catalog. It can format a well-formed call whose name does not exist. Applying the document is the later step that resolves catalog declarations, checks pins and types, plans graph changes, and reports whether the program can be represented.

FlowScript also carries comments with two very different jobs.

A normal body comment is prose:

```
eventsGeneric triageIncident(payload: Struct, report: string) {
    // Normalize before applying the incident rule.
    const normalized = report.trim()
}
```

The text parser and formatter preserve such a free-standing comment. A trailing prose comment may be normalized onto its own line. Today, however, these comments are not reconciled into durable Board comments, and top-level comments are skipped by the text AST. Treat this as a current round-trip gap, not as a documentation guarantee.

An anchor comment is identity metadata:

```
const urgentPhrase = "production is on hold"   //@v:variable-id

function normalizeReport(report: string): (normalized: string) {   //@l:layer-id
    return report.trim()
}

eventsGeneric triageIncident(payload: Struct, report: string) {   //@n:event-node-id
    const normalized = normalizeReport({ report: report })   //@n:call-node-id
}
```

`//@v:` identifies a Board variable, `//@l:` a Function layer, and `//@n:` a node. Anchors let an edit update the existing graph entity rather than guessing from a display name. They are emitted only when the authoring surface requests anchored text; clean examples in this book normally hide them.

Do not hand-edit an anchor casually. Duplicate anchors and stale anchors that cannot be rebound to one unique compatible entity are diagnostics; a safe unique rebind is reported as a correction. Removing anchored sections can produce a deletion plan that requires explicit approval. The comments may look small, but they are part of safe identity-preserving reconciliation.

6.2 `use` declarations

Every catalog operation has a namespace-aware FlowScript name. The fully qualified form is the least ambiguous:

```
log::error({ message: report, toast: false })
```

`::` separates a namespace path from a member. It is not field access on an object. A dot has a different role in expressions such as `report.trim()` and `incident.report`.

Top-level `use` declarations can open that namespace in four supported ways:

```
use log
use log::*
use log as audit
use log::{ error, info }
```

These forms allow, respectively:

```
log::info({ message: "qualified through imported namespace", toast: false })
info({ message: "bare through glob", toast: false })
audit::error({ message: "qualified through alias", toast: false })
error({ message: "bare selected member", toast: false })
```

The renderer does not add imports merely to make a file look busy. When a Board has at least two static call sites in one namespace and opening them will not collide with another name, current lowering can derive a glob import and render those calls bare. That is why the Chapter 4 Flow uses `use log::*` above `error(...)` and `info(...)`. Otherwise, keeping a qualified call can be the clearer canonical result.

Import resolution fails closed. An unknown namespace, a member that does not exist, a reserved alias, or two imports that expose the same name can produce a reconciliation diagnostic. If two globs expose the same member, argument shape may disambiguate a call; when it cannot, the diagnostic asks for a qualified spelling. An unused valid import is currently a non-blocking correction rather than a graph error.

`use` is top-level only. It changes how calls are resolved in this document; it does not install a package, grant a capability, or bypass App package approval. Namespaces and package governance are separate concerns.

6.3 Interfaces

An interface is the readable FlowScript surface of a structured schema.

Here is an incident contract:

```
interface Incident {
    report: string;
    source?: string | null = null;
    tags?: string[] = [];
}
```

`report` is required. `source` is optional and may be either a string or `null`; its default is `null`. `tags` is an optional array of strings with an empty default. The current interface surface also supports named types, maps, unions, literal string alternatives, `any`, and quoted field names when a schema property is not a valid identifier. Chapter 7 will treat those forms as a type system rather than a checklist.

Interfaces matter because "Struct" by itself says very little. A named interface can give a top-level variable, Function boundary, or Generic Event output an exact field contract:

```
let incident: Incident

function incidentReport(incident: Incident): (report: string) {
    return incident.report
}
```

The parser generates JSON Schema metadata from the declaration. The reconciler carries that schema onto the relevant variable or boundary pins and enforces it where the contract requires. When the Board already carries an equivalent schema, lowering can recover a readable nominal interface name instead of printing an opaque schema string beside every use.

This does not make `interface Incident` a runtime class. It has no constructor, prototype, or methods. It describes the shape moving through pins. A method-shaped call on an interface value still resolves to a catalog node or declared FlowScript function whose receiver contract matches that schema.

`@schema("…")` remains a legacy escape for object schema metadata that cannot be represented by a named interface. Prefer the interface form when it can preserve the schema. Do not manually copy a large JSON Schema into source merely because the syntax exists; the readable surface should earn its name.

One current boundary is worth recording: decorators on interfaces are not supported. An unused interface is also not an independent Board asset; interfaces are derived from schema-bearing text surfaces. Give the declaration a real variable, function, or event boundary to describe.

6.4 Top-level variables

Top-level variables are where JavaScript intuition becomes actively misleading.

Consider two declarations:

```
const urgentPhrase = "production is on hold"
let ignoreCase = true
```

At the top level, the keywords map to Board exposure:

FLOWSCRIPT	BOARD MEANING
`const`	A non-exposed Board variable
`let`	An exposed Board variable that may participate in App or Event configuration

They do **not** define runtime mutability. Both variables receive a fresh in-memory value for a run, initialized from a permitted runtime or Event override when the declaration enables that channel, and otherwise from the persisted Board default. Flow logic may assign either value during that run. The mutation is not durable across runs; persistent application state belongs in storage or Data Studio.

The distinction is easiest to remember this way:

> *At document scope, `const` and `let` describe configuration visibility. They do not import JavaScript's write rules.*

Annotations expose the remaining Board metadata. The current variable decorators are:

DECORATOR	MEANING
`@description("…")`	Human guidance for the variable
`@category("…")`	UI grouping metadata
`@secret`	Sensitive handling; the value stays outside rendered FlowScript
`@readonly`	User-editable metadata is false
`@runtime`	Configure the value through the runtime channel
`@schema("…")`	Legacy inline schema metadata when no interface represents it

Their canonical order is description, category, legacy schema, secret, readonly, then runtime. Each annotation applies to the declaration immediately below it; placing a comment between a decorator and its variable is currently a parse error.

The names can also invite the wrong conclusions. `@readonly` does not yet create a runtime write guard; it locks ordinary edits to that Board variable's definition and configuration. `@runtime` does not make a value global to the platform; it selects a runtime-configuration channel. Current Web and Desktop storage is local to the client profile and device, while an Event can carry its own trusted override. `@secret` does not permit credentials to be pasted into source. Chapter 11 examines those boundaries and the remaining enforcement gaps.

A secret declaration is intentionally value-free when rendered:

```
@description("Token used by the ticket integration")
@secret
@runtime
const ticketToken: string
```

FlowScript accepts an empty placeholder when creating such a declaration, but reconciliation rejects a non-empty secret initializer and does not echo that attempted value in its diagnostic. The real credential must enter through a trusted secret-setting surface. Existing secret defaults are omitted when the Board becomes text so viewing, copying, or sending FlowScript to an AI does not become a credential read channel.

This is separate from local bindings. Inside a body, `const normalized = report.trim()` names a node call result. A mutable `let` alias or a typed function-local variable belongs to that local scope. Those forms do not change whether a Board variable is exposed.

6.5 Functions

A FlowScript function is a callable Function layer with a typed boundary.

We can extract the normalization step from Incident Triage:

```
function normalizeReport(report: string): (normalized: string) {
    return report.trim()
}
```

The parameter becomes a typed input pin on the Function layer. The named return becomes a typed output pin. The body remains a graph inside the layer, and a call elsewhere becomes a Call Function node targeting that layer:

```
const normalized = normalizeReport({ report: report })
```

The declaration's return syntax is deliberately explicit:

```
: (normalized: string)
```

Parentheses hold a named output list, not a TypeScript tuple value. With several outputs, each name is part of the callable pin contract, and the `return` values must match the declared count and types. The reconciler reports unresolved or mismatched returns rather than inventing a wire.

Purity determines the execution boundary. The function above contains only demanded data work, so its layer does not need Execution pins. A function containing an impure call gains one execution entry and continuation, and every call joins the caller's execution path. This follows the same model as Chapter 5; `function` does not create a second runtime.

Within a body, a call-result binding normally renders with `const`:

```
const normalized = report.trim()
```

That is an SSA-like name for a node output, not the top-level non-exposed-variable rule. When a local `const name = expression` is merely aliasing a non-call expression, canonical rendering uses `let` instead. Scope matters more than the keyword's visual familiarity.

Functions may carry `@cache` metadata. The bare form uses the current default namespace, five-minute lifetime, and App scope; a structured form can set namespace, TTL, and App or user scope. A cache hit skips the function body, including any side effects, so caching belongs on logic whose output is truly determined by its input. We will treat that as an operational design choice rather than a formatting trick.

Function anchors use `//@l:` because the durable identity is the layer. The node statements inside still use `//@n:`. Renaming a function while preserving its layer anchor updates one callable entity; copying the body without its identity asks the reconciler to plan something new.

6.6 Events

Events come last because they are where declarations become runnable programs.

An event header has two names with different jobs:

```
eventsGeneric triageIncident(payload: Struct, report: string) {
    // body
}
```

`eventsGeneric` selects the catalog event type. `triageIncident` is the optional given name of this particular entry and becomes its readable identity. Without the second name, `eventsGeneric(...)` keeps the catalog's default naming.

The parameters are not ordinary function inputs. They are the event node's data output pins: the values made available when that entry starts. Generic Event can declare custom data outputs; fixed event kinds must use the outputs their catalog contracts provide. On an anchored existing

event, changing parameter name, order, type, container, or schema is treated as a boundary-contract change and is rejected rather than rewiring callers silently.

An event block is also distinct from an App Event. This block is the entry node inside the Flow. An App Event is the platform-level surface that may expose it as a form, API, schedule, quick action, or another supported trigger.

On a fresh Board, the Incident example can now use every document section we have learned:

```
use log::*

interface Incident {
    report: string;
    source?: string | null = null;
    tags?: string[] = [];
}

@description("Phrase that marks an urgent incident")
@category("Triage")
const urgentPhrase = "production is on hold"

@description("Whether the incident phrase ignores letter case")
let ignoreCase = true

function normalizeReport(report: string): (normalized: string) {
    return report.trim()
}

eventsGeneric triageIncident(payload: Struct, incident: Incident) {
    const normalized = normalizeReport({ report: incident.report })
    if (normalized.contains({ substring: urgentPhrase, ignoreCase: ignoreCase })) {
        error({ message: normalized, toast: false })
    } else {
        info({ message: normalized, toast: false })
    }
}
```

Read it from top to bottom. The glob import makes members of the `log` namespace available as bare calls; this example uses two of them. The interface gives the event a visible schema. The first Board variable is internal configuration; the second is exposed. The function names a reusable pure layer. The Generic Event supplies the payload and typed incident, calls the function, evaluates the rule, and selects one execution path.

Before the Board changes, the whole document is checked in phases. Syntax errors carry a one-based line and column. Reconciliation then resolves catalog and function declarations, validates argument and output

pins, checks types and schemas, plans execution wiring, validates function returns and event boundaries, and enforces structural limits.

The current backend can classify those findings with stable `FS_*` codes and phases such as parse, catalog resolution, type checking, execution wiring, lowering, and validation. Source spans for reconciliation findings are best effort because the structured diagnostic sidecar is derived from the existing string diagnostic channel; repeated call sites may be reported as a bounded set of candidate spans. Read the code, message, expected and actual values, declaration or pin, and safe repair guidance together instead of assuming every finding points to one perfect character.

Diagnostics are not permission to apply the parts that happened to work. The server treats a FlowScript document as one program: if reconciliation reports any diagnostic, no planned Board commands execute. Non-blocking normalizations, such as an unused import notice, are returned separately as corrections. A clean plan that removes existing Board entities still meets a separate deletion-approval gate.

That is the real anatomy of the file. Its order makes it readable; its anchors preserve identity; and its diagnostics protect the graph behind the text.

PART II · CHAPTER 07

Values, Types, Collections, and Interfaces

Learn how FlowScript types map to visible pins, including scalars, arrays, maps, sets, structs, interfaces, schema propagation, and type diagnostics.

FlowScript types are not annotations added to make source code look disciplined. They describe what may cross a visible connection.

That changes how we should learn them. A `string` is not only a compiler category; it is a string pin that can connect to another compatible pin. An `Incident` is not a hidden class; it is a structured contract whose fields can become visible on the Board. An array is not merely a value that happens to contain several things; its collection shape is part of the wire's contract.

The governing rule is simple:

> *If Flow-Like knows that two values are incompatible, it should reject the connection as early as possible. If an external system changes in a way the Flow could not know beforehand, execution should fail transparently at the node that first discovers the bad value and make the repair easy to find.*

APIs drift, models return unexpected content, and remote systems sometimes lie about their own schemas. Reliability means using every fact we have, then making the remaining uncertainty observable.

Release check: The scalar and container spellings, interface grammar, schema projection, connection diagnostics, and Make/Break behavior in this chapter are grounded in the current parser, renderer, reconciler, pin-matching code, and catalog tests. A publication release must still verify the complete examples in Studio and capture the exact diagnostic and jump-to-node experience. Schema support is evolving; the final section distinguishes today's readable surface from richer metadata that may remain behind it.

7.1 Scalars and `any`

A Flow-Like data pin has a base data type. FlowScript gives the current built-in types these canonical spellings:

FLOWSCRIPT	KIND OF VALUE	TYPICAL EXAMPLES
`string`	Text	names, messages, URLs, identifiers
`int`	Whole number	retry counts, status codes, quantities
`float`	Decimal number	scores, prices, measurements
`bool`	Boolean	enabled flags and decisions
`Date`	Date or timestamp	observation and scheduling times
`Path`	Flow-Like storage path	files and managed object-store locations
`bytes`	Binary data	encoded files, hashes, protocol payloads
`Struct`	Structured JSON-like value	API responses, records, configuration objects
`any`	Generic data value	a boundary whose concrete data type is not yet known

Execution pins are also typed internally, but they are control connections rather than values we store in variables. They answer "what runs next?" instead of "what data is this?"

The first four types have direct literal forms:

```
const service = "orders"
const retries = 3
const confidence = 0.95
const productionStopped = true
```

`Date`, `Path`, and `bytes` often arrive from catalog nodes or explicitly typed boundaries. A quoted timestamp is still a `string` merely because it resembles a date. The distinction matters: a Date node can offer date op-

erations, while a string can offer text operations. Conversion should be explicit when those contracts meet.

`bytes` represents one raw byte buffer on a Normal pin. `bytes[]` would be an array of buffers, not the usual representation of one file's contents.

`Struct` means "an object-shaped value," but it does not necessarily say which fields exist. That is useful at genuinely open boundaries, such as a webhook payload whose keys are selected by another system. It gives Flow-Like less information than a named interface, however.

`any` goes one step further. It is the textual spelling of a Generic pin: a value whose base type may specialize when it connects to something concrete. FlowScript permits this because some nodes really are generic. Array utilities, selectors, and external boundaries cannot always name a single element type in advance.

Permitted does not mean preferred. Flow-Like does not need an arbitrary style penalty for `any`; the workflow already makes the cost felt. With a typed value, compatible operations can be found by type, incompatible wires can be rejected, and fields can be exposed visibly. With `any`, the author often has to inspect, convert, or defer a decision until runtime.

The same is true for an untyped `Struct`. A schema-aware structure can use **Make Struct (Schema)** and **Break Struct** with visible field pins. An open structure pushes the author toward generic Get Field and Set Field operations with string keys. Both are possible. One is easier to build, read, and govern.

7.2 Value shapes

The base type is only half of a pin contract. Flow-Like also records the value's shape:

- **Normal** means one value.
- **Array** means an ordered collection.
- **Map** means values addressed by string keys.
- **Set** means a collection of unique values.

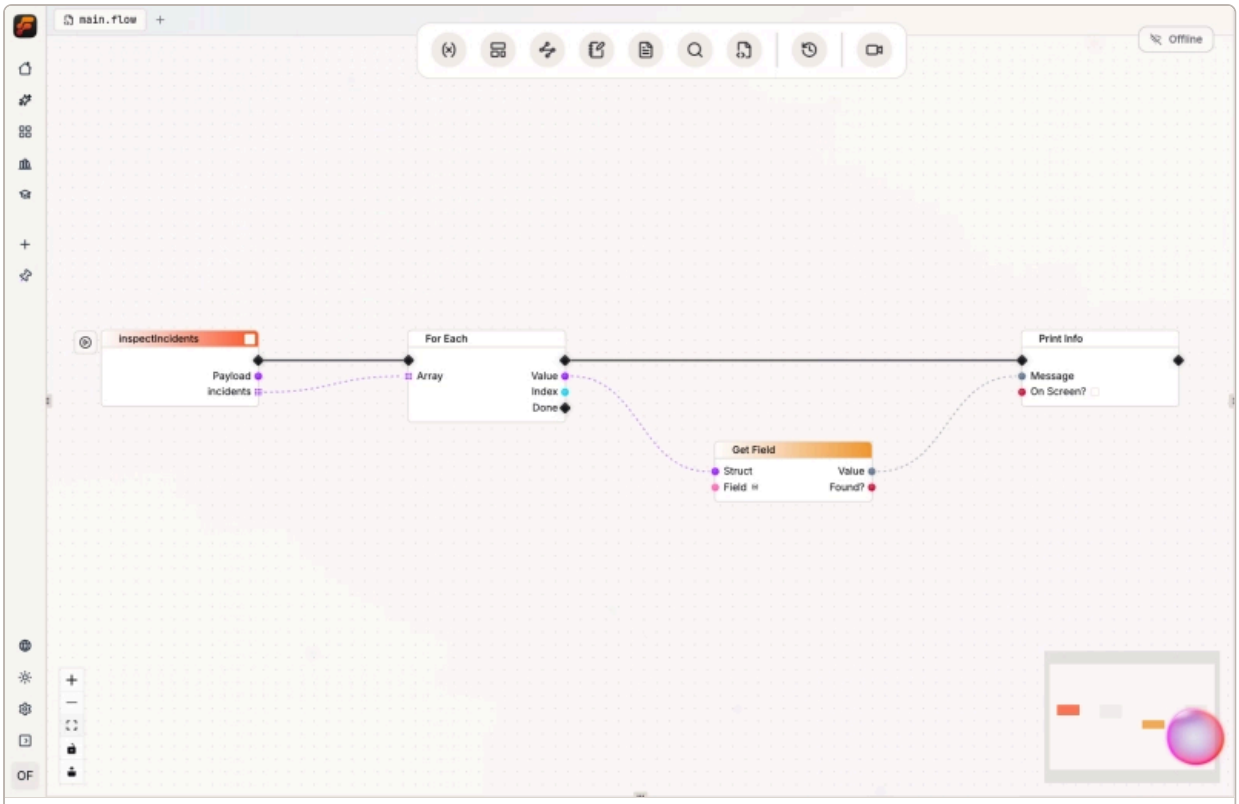
FlowScript renders those shapes in familiar syntax. Top-level Board variables may intentionally have no persisted default, as in this type-focused example:

```
const report: string
const reports: string[]
const incidentsById: Map<string, Struct>
const affectedServices: Set<string>
```

The canonical FlowScript map form uses `string` keys. The inner type names the values. A set has one element type. For ordinary declaration and function-boundary types, the current grammar wraps one base type in one collection shape; it is not an unrestricted TypeScript generic system.

Shape participates in compatibility. A `string` is not a `string[]`, and a `Struct` is not a `Struct[]`. Even Generic pins do not make that distinction meaningless: once a generic input or output declares a collection shape, current connection planning protects the scalar-versus- collection boundary rather than silently guessing.

This matters visually. Imagine a node that produces `Incident[]` and a node that accepts one `Incident`. The correct graph needs an operation that selects or iterates an element. This example makes the choice explicit with For Each:



The collection wire ends at For Each; only its per-item Value output crosses into the scalar Incident path.

Connecting the collection directly would hide which incident the consumer should receive. Rejecting the wire makes that decision explicit.

Array literals exist, but top-level defaults use compact canonical JSON, and an unannotated array default only establishes `any[]` today:

```
const names: any[] = ["api","billing"]
const typedNames: string[] = ["api","billing"]
```

The first declaration says “this is an array” without promising one element type. The second keeps the intended element contract visible. Empty arrays are even less informative, so an annotation is the honest choice when the type matters.

An object literal is a `Struct`, not automatically a `Map`. A struct has named fields that may each have different types. A map has arbitrary string keys whose values share one type. That difference decides whether we want named field pins or key-based lookup.

The current source surface has array and JSON object literals. Maps and sets are normally created and transformed through their catalog nodes rather than with JavaScript-style `new Map()` or `new Set()` expressions. That is consistent with the rest of FlowScript: construction remains a visible, declared node operation instead of arbitrary code hidden inside the source view.

7.3 Inference and its boundaries

FlowScript infers top-level variable types only where a literal provides an unambiguous answer. The present rules are deliberately small:

LITERAL	INFERRED DECLARATION TYPE
`"hold"`	`string`
`3`	`int`
`3.0`	`float`
`true` or `false`	`bool`
`{"system":"orders"}`	`Struct`
`[1,2]`	`any[]`
`null`	none; add an annotation

The renderer removes a redundant scalar annotation when the literal infers exactly the same type:

```
const retries: int = 3
```

becomes the canonical form:

```
const retries = 3
```

It retains an annotation when it carries information the literal does not. These declarations are not equivalent:

```
const threshold = 1
const thresholdAsFloat: float = 1
```

The first is an integer. The second deliberately declares a float despite using a whole-number default. Likewise, structured and array defaults keep their annotations in canonical source so their intended shape remains readable.

`null` cannot tell us whether the missing value will later be text, a date, an incident, or something else. FlowScript therefore rejects this:

```
const owner = null
```

and requires an explicit type. A top-level Board variable may omit its persisted default:

```
const owner: string
```

That is an unset default, not a declaration that every string pin accepts `null` as ordinary data. At interface-field level, nullability has a richer spelling—`string | null`—which we will use in the next section. Ordinary variable and boundary type references currently remain the simpler base-plus-container form.

Top-level inference is also limited to literals. FlowScript does not treat this as a hidden compile-time program:

```
const normalized = normalizeReport()
```

A top-level Board variable is persisted configuration, not the result of executing a node while the document loads. Node calls belong inside functions and events, where their output types come from catalog declarations and connected pins.

That separation gives Flow-Like two moments to catch a problem.

First, Studio checks candidate wires, and FlowScript reconciliation resolves calls against the live catalog. If a wired or anchored source contract says `int` and the target requires `string`, the new edge is rejected with a conversion diagnostic. Known shape and enforced-schema mismatches get the same treatment. Current Apply does not prove every direct literal against its target contract, so "known" must not be read as universal compile-time validation.

Second, execution validates reality. Suppose an external service used to return a customer ID as text and begins returning a number without updating any contract Flow-Like can see. No static checker can reject an unknown change. The first node whose typed operation cannot consume the value fails during the run. That may be a conversion or a later consumer if an open field lookup first produced `null`. Error and log evidence remain attributed to the discovering node, so the responder can jump to the operation that needs repair.

Early rejection and transparent runtime failure are not competing promises. Together they mean: catch everything knowable, and localize everything else.

7.4 Interfaces as visible schemas

A named interface turns an anonymous `Struct` into a readable field contract. Here is a more complete incident record:

```
interface AffectedSystem {
    name: string;
    region?: string | null = null;
}
interface IncidentRecord {
    id: string;
    severity: "critical" | "high" | "medium" | "low";
    report: string;
    source?: string | null = null;
    affectedSystems?: AffectedSystem[] = [];
    labels?: Map<string, string> = {};
    observedAt: Date;
    "external-ticket-id"?: string | null = null;
    externalPayload?: any = null;
}

eventsGeneric inspectIncident(payload: Struct, incident: IncidentRecord) {
    let report = incident.report
}
```

This one declaration carries several independent facts:

- `id`, `severity`, `report`, and `observedAt` are required fields.
- `source` may be absent, and when present may contain a string or `null`.
- `affectedSystems` is an array of another named structured type.
- `labels` is a string-keyed map whose values are strings.
- `severity` declares four literal alternatives in its schema rather than an unrestricted string.
- the external ticket key is quoted because its hyphens cannot be written as an identifier; and
- `externalPayload` admits uncertainty exactly where it enters, without weakening the rest of the record.

Optionality, nullability, and a default are different statements:

```
source?: string | null = null;
```

The `?` says the property is not required. `| null` says `null` is an allowed value when the property exists. `= null` records a default in the generated schema. A field can be required and nullable, or optional but non-null when supplied. FlowScript does not collapse those choices into one vague idea of “maybe.”

Inside interfaces, the current surface supports named types, nested arrays, string-keyed maps, unions, string-literal alternatives, `null`, and `any`. Parentheses disambiguate an array whose element is itself a union:

```
interface Evidence {
    observations?: (string | null)[] = [];
}
```

Without the parentheses, `string | null[]` would mean a string or an array of nulls, not an array whose individual elements can be either.

The parser turns an interface into JSON Schema metadata. When one interface references another, the generated schema carries the referenced definition. A variable, function boundary, or event boundary using

`IncidentRecord` therefore still becomes a Struct pin at runtime, but one with a specific schema attached.

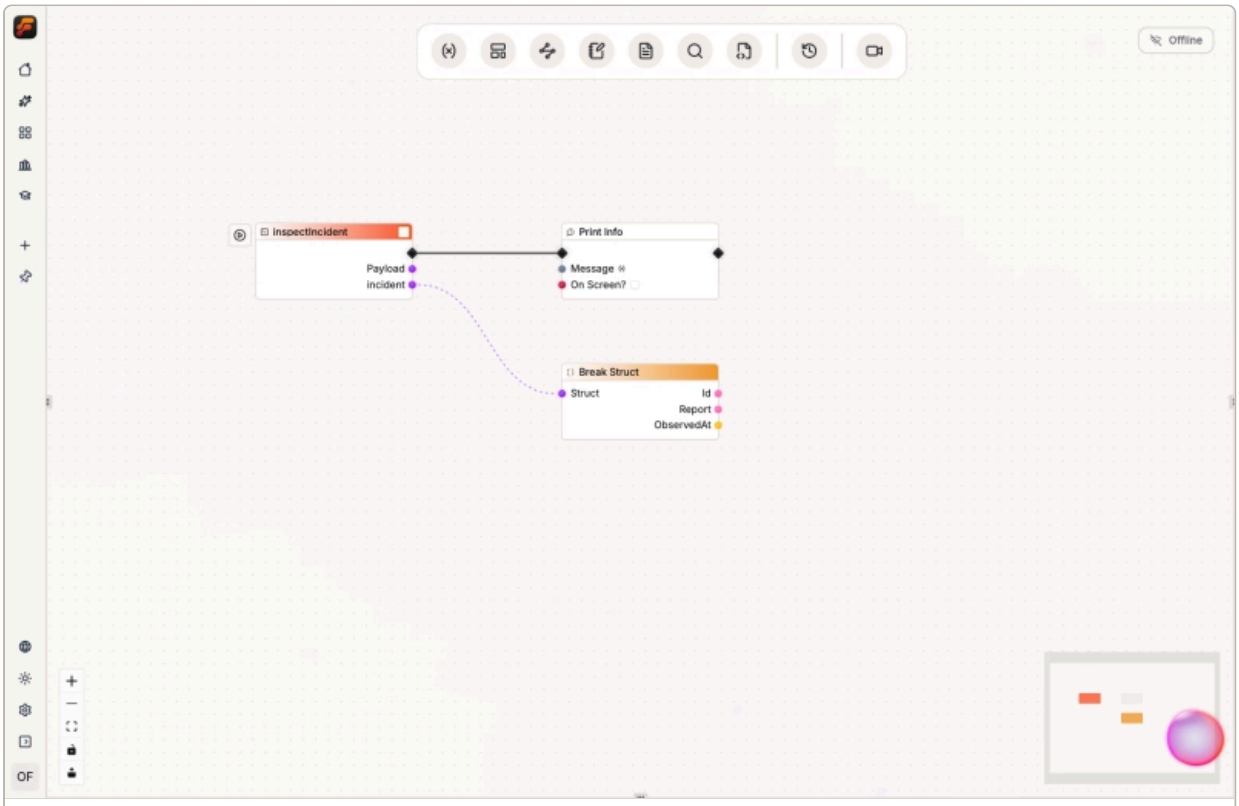
This is structural information, not object-oriented behavior. `IncidentRecord` has no constructor, prototype, or arbitrary methods. It describes fields on a wire. Method-shaped syntax still resolves to a catalog node or a FlowScript function with a compatible receiver.

7.5 Struct fields and schema propagation

The practical reward for typing a Struct appears on the canvas.

Given an open `Struct`, Flow-Like knows that the value is object-shaped but not which keys it contains. A generic Get Field node therefore needs both the Struct and a string such as `"report"`. A generic Set Field node needs the same kind of key. Rename the remote field and that string may remain plausible until a run reaches it.

Given `IncidentRecord`, the schema can drive field-shaped nodes instead:



Break Struct adopts the IncidentRecord schema and turns its three fields into visible, typed outputs.

Break Struct adopts the concrete schema from its producer and creates a field output pin with the most specific type it can currently derive for each property. **Make Struct (Schema)** does the inverse: it exposes field inputs derived from the consumer's schema and produces the structured value. Field selection becomes part of the visible graph rather than a bag of unverified string keys.

FlowScript can render that graph compactly:

```
let report = incident.report
let ticketId = incident["external-ticket-id"]
```

Plain identifier-like fields use dot syntax. Property names containing spaces, hyphens, or other characters that cannot form an identifier use bracketed string syntax. A numeric or dynamic bracket remains a collection index, so `incidents[0]` and `incident["external-ticket-id"]` keep different meanings.

The compact expression does not hide arbitrary reflection. Depending on the live graph, the renderer can be describing a Break Struct field output or a catalog field-access node. Applying the text must resolve that expression back to a compatible graph operation. Chapter 9 will follow that lowering in detail.

Schema propagation is what keeps the field information useful after the first wire. Current catalog behavior carries a typed array's item schema through common operations such as Get Element and For Each, so a downstream Break Struct can still expose the element's fields. Nested Struct fields receive standalone sub-schemas where the catalog can derive them. A generic passthrough must not overwrite a concrete schema with an "open object" marker.

Schema evolution should preserve the author's intent, too. The current Make/Break implementation does not silently delete a field pin while it still has a wire. If a new producer schema removes the wired `name` field, the stale pin and its connection remain, the new fields are added, and the node receives an error naming what is still connected but no longer declared. The author can jump to that node, insert a conversion or select the renamed field, and repair the Flow without first reconstructing a vanished wire.

That is the concrete answer to “what happens when an interface changes?” There is no single rule based only on whether a field was added, removed, or renamed. Flow-Like uses the information available at each boundary:

- a new optional field may require no repair at an open or schema-adopting boundary, although two enforced whole-schema contracts can still compare unequal;
- a known type or schema contradiction is rejected before a new edge is applied;
- a removed field that is still wired is retained and reported rather than silently erased; and
- an unannounced external change can only be discovered at runtime, where the first node unable to consume the changed value exposes it.

The principle matters more than pretending every provider follows perfect versioning: catch the change wherever it becomes knowable, and take the responder directly to the place that must convert or adapt it.

7.6 Edges and preserved metadata

Types protect edges, but the protection is intentionally based on facts—not wishful inference. The following matrix is a useful working model:

WHAT FLOW-LIKE KNOWS	CURRENT BEHAVIOR
Known source and target contracts have different base types, such as `int` and `string`	Reject the new connection and request an explicit conversion
Scalar versus collection shape	Reject where the declared shapes contradict one another
Two incompatible concrete schema contracts	Reject when the pins require those contracts to be enforced
A direct literal has the wrong input type	Studio may warn, but Apply does not yet validate every literal; the consumer may reject it at runtime
One side is genuinely Generic or declares an open Struct shape	Permit specialization or schema adoption where that node contract allows it
A typed Make/Break field disappears while still wired	Preserve the wire and report the stale field on the node
An external value violates a fact the Board did not know	Fail at runtime with node-attributed error evidence

`any` is therefore an escape from a fact we do not possess, not an escape from every collection rule. An open Struct schema similarly says "these fields are not fixed"; it must not be mistaken for a different concrete object schema. Make and Break boundary pins are special because they adopt the connected Struct schema dynamically.

Schema compatibility is not currently a full structural-subtyping theorem. Where two pins enforce concrete schemas, the safe comparison is generally canonical contract equality, with explicit schema-adopting boundaries as exceptions. Authors should not assume that a record with three fields can always flow into a two-field input merely because it looks like a TypeScript subtype. The node declaration decides whether its schema is descriptive, enforced, or open.

The Board stores schema metadata as JSON Schema text or as a reference to it. FlowScript interfaces project the most readable portion of that metadata into source. Today that readable surface handles object properties, requiredness, supported defaults, named definitions, arrays, string-keyed maps, supported unions and literal strings, `null`, `any`, and date/date-time formats.

Some JSON Schema carries more than this grammar can say: validation bounds, titles and descriptions at every level, pattern properties, or compositions such as `oneOf` and `allOf` may exceed the readable interface projection. The legacy `@schema("…")` decorator remains an escape hatch for an object schema that cannot be represented as a named interface. It is exact, but it is not pleasant prose and should not be the first choice for an ordinary record.

There is an important preservation nuance. When a Board already contains a richer live schema, the renderer can show its representable interface projection while the reconciler retains the exact schema on an unchanged Board-to-source-to-Board round trip. The visible interface does not necessarily prove that every hidden constraint could be recreated from a new standalone text file. Preserving known Board metadata and authoring every possible JSON Schema feature are different capabilities.

Collection surfaces have release boundaries as well. Top-level and function/event type references currently support Normal, array, `Map<string, T>`, and `Set<T>` shapes around a single base type. Interface fields support richer nested arrays, maps, and unions, but `Set<T>` is not yet part of

the interface-field grammar. Current Make/Break field inference is narrower still: an enum-only literal field can materialize as Generic, while a map-shaped property can materialize as an object-shaped Struct. The root interface contract remains available, but those derived field-pin types are not yet as precise as the source. Treat these facts as release observations, not as a manifesto against future type-system growth.

Named-type lookup has another current edge: a top-level, function, or event type name that does not resolve to a declared interface can fall back to a schema-less Struct. A misspelled interface name is therefore not guaranteed to be rejected today. Keep publication examples under catalog-aware reconciliation tests instead of treating parser acceptance as nominal type safety.

For publication, any example near these edges should pass three checks against the named release:

1. parse and render the FlowScript;
2. apply it with the release's catalog and inspect the resulting Board pins and schemas; and
3. render that Board back to FlowScript and compare the meaningful contract.

If a shape fails one of those checks, report the smallest reproducible source, the release, the expected field or container metadata, and the actual result. Do not turn one unsupported round trip into folklore that "FlowScript can never support this." The language is evolving, and honest release notes are more useful than permanent-sounding limitations.

Types in FlowScript serve one purpose across both views: make the workflow easier to construct, harder to connect incorrectly, and faster to repair when reality changes. The next chapter uses those contracts to navigate the catalog—the large, typed node library that gives FlowScript its real capabilities.

PART II · CHAPTER 08

Calling the Node Library

Discover and call Flow-Like catalog nodes with qualified names, imports, method syntax, named output pins, generated declarations, and typed search.

The useful size of a programming language is not measured only by how many keywords it has. It is measured by how much real work we can express without leaving its model.

FlowScript keeps its own syntax deliberately small and puts most capabilities in the node catalog. Text operations, HTTP requests, files, databases, document processing, models, automation, UI, and many other domains arrive as typed operations. Built-in nodes and nodes from packages use the same call surface. In Studio they are boxes with pins. In FlowScript they look like functions and methods. At runtime they remain the same named operations.

That is why the catalog is best understood as FlowScript's expansive standard library, with one important qualification: it is not a pile of arbitrary functions linked invisibly into a process. Every call resolves to a catalog declaration and therefore to a node the Board, runtime, and run evidence can identify.

The practical goal of this chapter is not to memorize that library. It is to learn how to ask it a precise question: *given this value and this intent, which compatible node can do the next piece of work?*

Release check: Qualified calls, all four `use` forms, receiver dispatch, name-based output destructuring, generated declarations, editor completion, and context-sensitive Studio search are implemented in the current repository. Safety-, trust-, permission-, performance-, and cost-aware result ranking is product direction, not current catalog ordering. Versioned node migration also has a gap described in Section 8.5: today's general schema synchronizer can remove stale pins or detach changed-type pins without retaining an author-facing node error.

8.1 Qualified calls

The most reliable way to read a node call is the fully qualified form:

```
namespace::operation({ pinName: value })
```

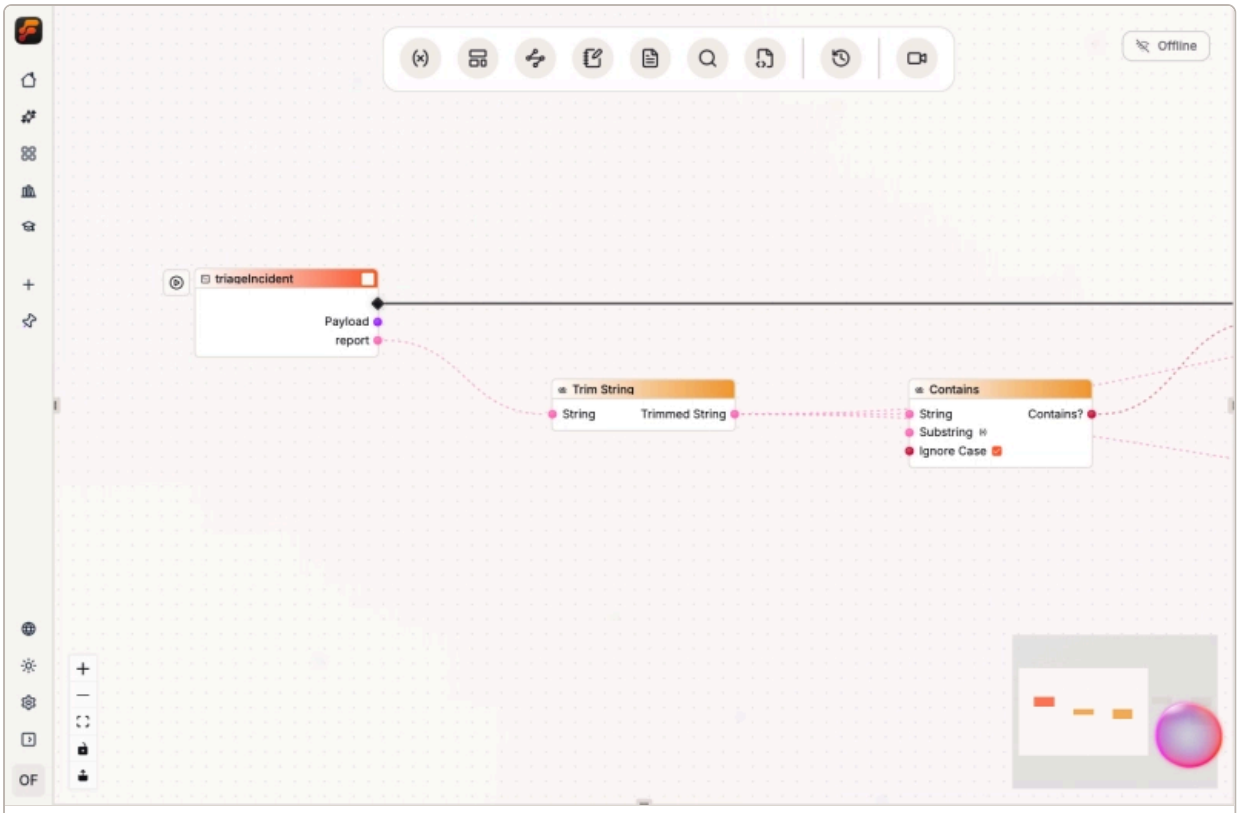
The path identifies a catalog namespace and member. The object maps values to input pins. Consider the first operation in our Incident Triage Flow:

```
const normalized = string::trim({ string: report })
```

`string` is the namespace, `trim` is the FlowScript alias of the Trim String node, and `string` in the argument object is that node's input pin. `normalized` receives the node's sole data output. Nothing in this expression executes an arbitrary function named `trim`; reconciliation looks up the live catalog declaration, plans the node, and wires `report` to its declared input.

Catalog pins often use snake_case internally. Their FlowScript names are rendered in camel case, so a pin called `ignore_case` appears as `ignoreCase`. This is a stable textual projection of the pin name, not a new free-form parameter. Completion inserts the correct spelling, and Apply can report an unknown argument, a missing required input, a duplicated input, or a call that does not match a declaration.

Named arguments are worth keeping as the canonical mental model even where compact positional forms are accepted. They make the graph mapping visible in the source:



The named string argument becomes the visible input pin on Trim String; its result remains an ordinary typed output wire.

They also make reviews resilient to a long signature. Someone reading an HTTP call should not have to remember whether the fourth Boolean controls redirects, certificate validation, or retries. The pin name carries that meaning beside the value.

An input with a catalog default is optional in the generated declaration and may be omitted. A required input has to be provided by a literal, an expression, or a wire-producing binding. The declaration is the authority; TypeScript familiarity is not permission to invent an option that a node does not expose.

`::` is significant. It joins namespace segments, as in `ai::ml::read`. A dot means something is being selected from or invoked on a value. The reconciler currently recognizes some mistaken `string.trim(...)` namespace calls and suggests `string::trim(...)`, but relying on that correction would blur two different ideas. Write `::` for a namespace and `.` for a receiver or output.

Here is the chapter's Incident example. It intentionally uses several equivalent call surfaces so we can examine them in the following sections:

```
use log::{ info, warn }
use string as text

eventsGeneric catalogIncident(payload: Struct, report: string) {
    const normalized = text::trim({ string: report })
    const { before: system, after: detail, found } = normalized.splitOnce({ separator: ":", fromEnd: false })
    if (found) {
        info({ message: detail, toast: false })
    } else {
        warn({ message: normalized, toast: false })
    }
}
```

The Flow receives a report such as `orders: production is on hold`, trims it, attempts to split a system name from the detail, and records whether that structure was found. The example adds no hidden parser routine. Trim, Split Once, Branch, and the log operations remain visible nodes.

A qualified call does not install a package, grant a permission, or make an unavailable node exist. It can only resolve against the catalog available to the App and release. That boundary is essential: names make capabilities addressable; App packages and governance decide which capabilities are present. If the relevant package is absent, Apply reports an unknown namespace, member, or catalog declaration instead of placing a pretend node.

8.2 Imports and aliases

Repeating long namespace paths can obscure the actual logic. Top-level `use` declarations change how names are resolved within one FlowScript document. They do not change the nodes placed on the Board.

FlowScript supports four forms:

```
use ai::response
use string::*
use string as text
use log::{ info, warn }
```

Each form opens a different scope:

FORM	MEANING	EXAMPLE CALL
`use ai::response`	Bring the final namespace segment into scope	`response::make({ ... })`

FORM	MEANING	EXAMPLE CALL
`use string::*`	Make every member of one namespace callable bare	`trim({ string: report })`
`use string as text`	Give the namespace a local alias	`text::trim({ string: report })`
`use log::{ info, warn }`	Make only selected members callable bare	`warn({ message: report })`

Our example aliases `string` as `text` and imports two logging members. The alias is local vocabulary, not a fork of the catalog. `text::trim` and `string::trim` still resolve to the same node type. Likewise, bare `warn(...)` is not a user-defined function merely because its namespace is absent from the line.

Imports are intentionally checked. An unknown namespace, a nonexistent member, an alias that is a FlowScript keyword, or selected members that collide can produce a diagnostic. A valid but unused import is currently reported as a non-blocking correction. If two opened namespaces export the same member, reconciliation first considers whether the supplied argument shape identifies one candidate. When it cannot decide safely, it asks for a qualified spelling rather than choosing by catalog order.

That last rule matters in a large library. Both arrays and strings can reasonably offer an operation named `contains`. A bare call may be clear when its named inputs fit only one declaration. `string::contains(...)` is the unambiguous escape whenever the context is not enough. User-declared functions also own their names and can shadow bare catalog members or method aliases; the diagnostic explains the conflict instead of silently changing which operation runs.

Imports are not durable Board entities. The renderer derives them again from the calls it sees. Current lowering opens a namespace with `use ns::*` when that namespace has at least two static call sites and none of its used member names collide with a Function, another placed node's flat name, or a member already opened from another namespace. Method-form calls do not count as static sites. A single call normally stays qualified.

As a result, source may re-render with a different but equivalent import style after Apply. An author-written alias such as `text` is useful while editing, but the Board stores the resolved node identity, not the alias. The canonical renderer may later choose `string::trim` or derive `use string::*`. That is healthy normalization, provided the resolved graph is unchanged.

The simplest style rule is therefore:

- begin with qualified calls when learning or resolving ambiguity;
- accept autocomplete's import edit when a repeated namespace is making the code noisy; and
- let canonical rendering decide whether the final Board still earns that import.

8.3 Method form

Some catalog nodes designate one data input as a receiver. Their generated declaration contains a `this:` parameter. Split Once currently declares its contract in this shape:

```
function splitOnce(
    this: string,
    { string: string, separator: string, fromEnd?: bool }
): { before: string, after: string, found: bool };
```

The `this: string` marker says the input pin named `string` may be supplied by the value to the left of a dot. These two calls resolve to the same node:

```
const split = string::splitOnce({
    string: normalized,
    separator: ":",
    fromEnd: false,
})

const splitAgain = normalized.splitOnce({
    separator: ":",
    fromEnd: false,
})
```

In method form, the receiver pin is already bound. Supplying `string: normalized` again is an error because it would give one pin two sources. When one required input remains, it can be positional: `normalized.splitOnce(":")`. With several inputs or meaningful options, the named object remains clearer.

This is not JavaScript prototype extension. A string does not acquire arbitrary methods, and `splitOnce` is not implementation hidden inside a string object. The call still becomes a Split Once node with an ordinary String input wire. Its duration, failure, permissions, and outputs stay attributable to that node.

The receiver contract also powers discovery. After a value known to be `string`, the current FlowScript editor offers methods whose receiver class is string, plus genuinely universal operations. After an array it offers array operations; after a titled Struct it can include methods for that schema and general Struct methods. Completion uses the active catalog, so nodes from available packages can participate in the same table.

If the receiver type is unknown, completion becomes broader and method resolution may have several candidates. An `any` value followed by `.contains(...)` could mean a string operation, an array operation, or another package declaration. FlowScript checks argument shape and opened namespaces where it can, but it will demand a qualified call when candidates remain tied. This is another place where the ergonomic cost of `any` appears without banning it.

Receiver metadata, rather than namespace spelling alone, decides whether a node has method form. A hash operation can therefore declare its text or byte input as a receiver even though its static name lives under `hash`. Conversely, a node author can opt out when a method spelling would be misleading. The generated declaration is where readers should verify the decision.

FlowScript Functions can also be invoked in a method-shaped style through their first parameter. That convenience is still a call to a visible Function layer, not permission to attach an opaque body to a value. We will return to Function contracts later; for catalog calls, the rule remains: the dot binds one declared input pin.

8.4 Single and multiple outputs

A node can have no data outputs, one data output, or several. FlowScript reflects each shape without turning output order into an invisible convention.

A node with one data output behaves like a value-producing function:

```
const normalized = text::trim({ string: report })
```

`normalized` refers to Trim's only data output. An impure logging node has no data output, so it is normally written as a statement:

```
warn({ message: normalized, toast: false })
```

For several data outputs, select them by pin name. Object destructuring makes all selected wires clear at once:

```
const { before: system, after: detail, found } = normalized.splitOnce({
    separator: ":",
    fromEnd: false,
})
```

The left side before the colon is the output pin. The right side is the local binding. `before` becomes `system`, `after` becomes `detail`, and `found` keeps its own name. If a release changes an output name, reconciliation can identify the missing pin instead of quietly binding "the second output" to a different value.

The alternative is to retain a name for the call and select outputs as fields:

```
const split = normalized.splitOnce({ separator: ":", fromEnd: false })
if (split.found) {
    warn({ message: split.after, toast: false })
}
```

Here `.found` and `.after` select output pins on the Split Once node. This looks like object field access but resolves first against the node's declared outputs. Chapter 9 will show how the same surface distinguishes an out-

put selection from a Struct field read.

FlowScript recognizes a conventional default output when a node has several data outputs. Current resolution uses a sole output automatically and, among several outputs, looks for names such as `result`, `value`, `output`, `out`, or `batch`. That lets a call serve directly as a value while its other pins remain selectable. Split Once has `before`, `after`, and `found`, but no such default, so code that consumes one result must name it explicitly.

Array destructuring is intentionally unsupported:

```
// Rejected: output meaning must not depend on pin position.
const [system, detail, found] = normalized.splitOnce(":")
```

Pin order is presentation metadata that can change as a node evolves. Pin names are the contract. Object destructuring makes that contract survive reordering and makes the corresponding three data wires obvious on the Board.

The same principle applies when one output feeds several consumers. Binding a local does not copy the value into hidden language state; each use resolves to the relevant output pin and the graph records the fan-out. FlowScript gives that wire a readable name while preserving its source.

8.5 Generated `.flow.d` declarations

Editor assistance is only trustworthy when it describes the catalog that will reconcile and run the Flow. Flow-Like generates `.flow.d` files from node metadata to create that bridge.

A declaration for Contains looks like this, abbreviated only by removing prose lines:

```
declare namespace string {
    /**
     * @node string_contains @receiver string @alias stringContains
     * @param string — receiver in `x.contains(...)`
     * @param substring
     * @param ignoreCase (optional)
     * @returns contains
     */
    function contains(
        this: string,
        { string: string, substring: string, ignoreCase?: bool }
    ): bool;
}
```

Each part has a job:

- the namespace and member provide the qualified spelling;
- the object lists the static call's complete data-input shape;
- `?` records an optional input;
- `this:` and `@receiver` identify method binding;
- the return type describes one output or an object of named outputs;
- `@node` preserves the internal catalog identity;
- `@alias` preserves the legacy flat camelCase spelling; and
- `@impure`, when present, records that the node has Execution pins.

Descriptions for the node, pins, and outputs become hover and declaration documentation. Struct schemas live in a generated sidecar so tooling can resolve nested fields without expanding large JSON Schemas into every human-readable signature. A `names.json` sidecar maps internal node types, qualified names, aliases, receiver pins, receiver classes, and categories.

The generated files are grouped both by top-level domain and, under a package index, by source package. That serves several consumers from one contract: Studio's FlowScript editor builds completion, hover, signature help, and diagnostics from the active catalog; the VS Code extension can read declaration signatures; and FlowPilot can query the declaration index by intent or exact spelling before it writes source. Humans and AI therefore receive the same pin names instead of maintaining separate handwritten API summaries.

Generated does not mean timeless. A declaration snapshot must match the catalog available to the App. Package additions can introduce new nodes or collisions, and node upgrades can change pins. The syntax parser does not bind calls by reading `.flow.d`; the live catalog used during reconciliation is the semantic authority. Examples in this book therefore need catalog-aware reconciliation, not merely a successful parse.

The current Board model gives nodes a schema version for migration. When the catalog version is newer, general synchronization matches pins by name. It preserves pin IDs and wires when the declared base type and collection shape remain compatible, adds new pins, refreshes catalog-owned metadata, and gives dynamic nodes a chance to rebuild their derived pins. Widening a data pin to Generic also preserves an existing compatible wire.

The founder's intended rule is stronger: migrate safely where possible; where safety cannot be proved, keep the node in place and annotate the problem for explicit repair. The node itself does remain on the Board today, but the general synchronizer does not yet fulfill the second half uniformly. On an ordinary node it removes a pin absent from the newer catalog. When a matched pin's type or collection shape changes, it clears that pin's connections and resets its default. It then clears the node error. Some schema-driven nodes already do better by retaining still-wired stale pins and placing an error on the node, but that behavior is not yet the universal upgrade contract.

There is one more useful preservation path today. If an entire package or node type is no longer available, Studio keeps the placed node on the Board and marks it with an unavailable-package warning. That protects the location and identity of the missing capability. It does not repair a pin-level breaking change inside a package that is still present, which is why the broader migration contract still matters.

This is a release gap, not wording to conceal. Until migration follows the intended rule everywhere, inspect package-update changes and the affected Boards before treating an upgrade as automatic. The durable product direction is that an unsafe change must become a visible repair, never a silently different program.

8.6 A large library without a memory test

The catalog should reward knowing the problem, not knowing 1,668 function names.

That number is a snapshot, not a product slogan: the checked-in generated declarations contain 1,668 node entries at repository revision `839395640`. Packages and releases change it. More important than the count is that the library exposes enough type and intent metadata to make discovery contextual.

Studio offers two complementary paths. Opening the Actions catalog on an empty part of the canvas lets an author browse categories or search across node names, friendly names, categories, descriptions, and input and output pin names. Current text search supports prefixes and a small amount of fuzzy matching. Without a query, nodes and categories are primarily ordered by name.

Dragging a wire from a pin into empty space adds the stronger clue: its **Context Sensitive** mode is enabled by default. The catalog keeps operations that have an opposite-direction pin compatible with the value being carried. Search then narrows that compatible set, and choosing a node connects the matching pin. An author holding a String output does not need to browse every database writer, event source, and image operation before finding Trim or Split Once.

Compatibility here is substantive. The current filter considers direction, data type, collection shape, Generic specialization, schema enforcement, and the special schema-adopting Struct boundaries. The actual connection still passes through Board validation. Turning Context Sensitive off is possible when deliberately exploring the whole catalog, but the default workflow makes the typed path the shortest one.

FlowScript applies the same idea at the cursor. After `normalized.` the editor can infer a String receiver and show the node methods available for that class. After `string::` it lists namespace members. At a bare call site it can offer an unopened member together with an edit that inserts or extends the appropriate `use` line. Hover reveals the signature; named-argument completion offers remaining pins; enum-constrained pins can offer

their allowed values. FlowPilot's declaration lookup supplies the AI-facing equivalent: search by intent, then return exact live signatures and call forms instead of asking the model to guess.

There is still room to improve the order of correct answers. Current visual search ranks textual relevance using boosted name, friendly-name, category, pin, and description fields. It does not currently rank compatible candidates by package trust, permission breadth, node quality scores, expected performance, or cost.

The desired direction is a policy-aware ranking after compatibility. If two nodes both solve the task, Flow-Like should be able to prefer the one that better fits the organization's trust and permission rules and its performance and cost priorities. That ranking should explain itself—such as "built in, no network permission, lower expected cost"—and keep alternatives inspectable. The exact score direction, weighting, override policy, and provenance still need to be settled in the governance work; calling this current behavior would be premature.

Discovery therefore forms one continuous loop across the two views:

```
Start with a typed value or required pin
    → filter to compatible operations
    → search by the task you mean
    → inspect the declaration and tradeoffs
    → place or call the node
    → let reconciliation verify the exact contract
```

When no catalog operation expresses the intent, the answer is not an inline general-purpose code block. Compose lower-level visible nodes, or add a reviewed package whose node exposes a typed, capability-scoped contract. Later chapters will show how those WASM extensions enter an App. The important boundary is already visible: new power joins the same searchable catalog and the same graph, instead of opening an opaque hole in it.

The result is a language with a small surface and a large reach. Qualified calls give us an exact address. Imports remove repetition. Method form turns a known type into a discovery tool. Named outputs keep wires sta-

ble. Generated declarations let people, editors, and AI work from one node contract. We do not memorize the library; we navigate it through types and intent.

The next chapter looks beneath the most familiar-looking FlowScript expressions. Operators, templates, ternaries, and field access are convenient syntax, but each must still lower honestly to catalog operations and visible wiring.

PART II · CHAPTER 09

Expressions, Operators, and Readable Sugar

See how FlowScript operators, ternaries, template literals, and struct field access lower to visible catalog nodes without hiding configuration or data flow.

FlowScript is at its most familiar when the nodes seem to disappear:

```
let urgent = production && (attempts ⩾ 3)
let status = urgent ? "critical" : "investigate"
let summary = `${status}: ${report}`
incident.status = status
```

This looks like ordinary code. It is also a compact description of a graph containing an Integer Greater Than or Equal node, Boolean And, Select, String Format, and Set Field. Apply does not pass these expressions to a hidden JavaScript engine. It resolves each one against the catalog and creates or updates the corresponding nodes, pins, values, and wires.

That distinction gives us the rule for every convenience in this chapter:

> *FlowScript may make a Flow shorter to read, but it may not make the Flow less exact.*

If a short form can reproduce the complete node contract, the renderer may use it. If meaningful configuration would disappear, the explicit call remains. Readability is valuable because it helps people reason about the system; it is not permission to conceal a decision.

Release check: Arithmetic, comparison and Boolean lowering, unary `!` and `-`, the four compound assignments, value-selecting ternaries, template literals, and Struct field writes are implemented in the current repository. Mixed known operand types are rejected. The current diagnostic does not yet offer the two intent-specific conversion repairs proposed in Section 9.1. Operator and template rendering already preserve configured extra pins in the cases described below, but three current asymmetries remain: Float equality and inequality can render as operators that text cannot reconcile, Integer division's declared operator result disagrees with its live node output, and a Struct Get `found` output can be mistaken for field-value sugar. Those are implementation gaps, not part of the language contract.

9.1 Arithmetic, comparison, and Boolean expressions

An operator is a catalog lookup with two extra clues: the symbol and the operand type. Given:

```
let urgent = attempts ⩾ 3
```

FlowScript knows that `attempts` is an Integer input and `3` is an Integer literal. It can therefore select the Integer Greater Than or Equal node and connect its Boolean output to the next consumer. The source spelling is compact; the Board still shows the operation that made `urgent`.

The current operator families are intentionally specific:

VALUES	SHORT FORMS THAT CURRENTLY LOWER TO CATALOG NODES
Integer	`==`, `≠`, `>`, `⩾`, `<`, `⩽`, `+`, `-`, `*`, `%`, `**`
Float	`>`, `⩾`, `<`, `⩽`, `+`, `-`, `*`, `/`, `**`
String	`==`, `≠`, `+`
Boolean	`==`, `&&`, `||`, `^`

The table is narrower than the parser's vocabulary. A token being syntactically recognizable does not prove that the active catalog contains an unambiguous node contract for it. Apply is where the symbol, types, and live catalog meet.

The parser also recognizes `|`, but current reconciliation has no operator node mapping for it. `===` and `!==` are accepted compatibility spellings and normalize semantically to `==` and `≠`; FlowScript does not define a separate JavaScript-style identity comparison. Integer `/` is omitted from

the table because the current operator registry expects an Integer result while the live Integer Divide node returns a Float. Until those declarations agree, use the explicit catalog operation and inspect its Float output rather than relying on `int / int` sugar.

Float equality and inequality demonstrate why that matters. Comparing floating-point values usually needs a tolerance. Flow-Like's Float Equal node exposes that tolerance as an input, so `a == b` cannot honestly express its full decision. Use the explicit call and choose the tolerance:

```
const closeEnough = float::equal({
    float1: measured,
    float2: expected,
    tolerance: 0.001,
})
```

The longer form is better here. It puts an operationally meaningful assumption in the source and on the node instead of inheriting an invisible global default.

Precedence is familiar; canonical source is explicit

FlowScript reads binary expressions with the usual precedence groups. From low to high they are Boolean OR, Boolean AND, `|`, then `^`, equality, ordering comparisons, addition and subtraction, multiplication/division/modulo, and exponentiation. Exponentiation associates to the right; the others associate to the left.

The parser can therefore understand:

```
let urgent = production && attempts ⩾ 3 || manuallyEscalated
```

The canonical renderer adds parentheses around nested binary expressions:

```
let urgent = (production && (attempts ⩾ 3)) || manuallyEscalated
```

That is not a semantic change. It makes the tree—and consequently the chain of operator nodes—obvious in a review. Parentheses cost very little compared with an incident caused by two readers remembering prece-

dence differently.

Types decide; FlowScript does not guess intent

The same `+` symbol can identify Integer Add, Float Add, or String Concat. FlowScript chooses only when the operand evidence gives it one meaning. Two strings concatenate. Two integers add. Two floats add. An Integer literal may adopt the Float family when the other operand is known to be a Float, which makes `ratio * 2` natural. Two separately typed Integer and Float values remain different contracts and require an explicit conversion.

Consider the deceptively small expression:

```
let result = "5" + 1
```

There are at least two plausible results:

- numeric addition after parsing the left side: `6`; or
- text concatenation after formatting the right side: `"51"`.

Silently inserting a conversion would make the program compile by choosing a business decision the author never made. Current reconciliation instead reports incompatible `String` and `Integer` operands and creates no String Concat node. Apply is atomic: a reconciliation diagnostic prevents the edit commands from being applied, so the failed expression cannot leave half of a repair on the Board.

The right repair depends on intent. Numeric intent can be made explicit with a typed parser and its success output:

```
const { integer: parsed, success } = "5".toInt({ fallback: 0 })
if (!success) {
    // Handle invalid input as part of this Flow's domain policy.
}
let total = parsed + 1
```

Text intent can be made explicit with formatting:

```
let text = `5${1}`
```

The desired editor experience is: reject the ambiguous Apply, offer both repairs, preview the node each would insert, and let the author submit a second Apply for the chosen repair. Canonical source then rewrites to that visible choice. The editor should not choose one automatically. A conversion can fail, truncate, select a fallback, or alter representation; its policy belongs in the Flow.

The same principle applies to `any`. An `any + 1` expression has no reliable operator family. Narrow it through a typed interface, use a typed parser such as `string::toInt`, or place `types::tryTransform` and consume its `success` result. Try Transform is a real catalog node whose Generic output adapts to its typed consumer; it is not an invisible coercion rule attached to every operator. An untyped or unconnected Try Transform output produces `null` and `success = false`; a failed conversion is an ordinary result rather than a node error. Ignoring `success` merely moves the eventual failure to a later typed consumer.

9.2 Unary and compound assignment

FlowScript supports two prefix conveniences:

```
let unavailable = !healthy
let debt = -balance
```

Boolean negation becomes the Boolean Not node. Unlike binary operators, canonical source will generally show that catalog call rather than preserve `!healthy`. Numeric negation canonicalizes to subtraction from zero:

```
let debt = 0 - balance
```

The reconciler then chooses Integer Subtract or Float Subtract from `balance`'s type. A negative literal such as `-3` remains a literal; an expression such as `-balance` becomes an operation. This small difference is visible on the Board because one needs a node and the other does not.

Four compound assignments are accepted while writing:

```
attempts += 1
remaining -= consumed
scale *= 2.0
scale /= 4.0
```

Canonical rendering expands them:

```
attempts = attempts + 1
remaining = remaining - consumed
scale = scale * 2.0
scale = scale / 4.0
```

This expansion exposes the actual model. The right side reads the current producer, the operator creates the next value, and the name is rebound for later statements. There is no opaque CPU-style `+=` instruction inside the graph. The compact input is accepted for convenience; the durable text makes the read, operation, and rebind explicit.

Field compound assignment follows the same rule:

```
incident.attempts += 1
```

becomes:

```
incident.attempts = incident.attempts + 1
```

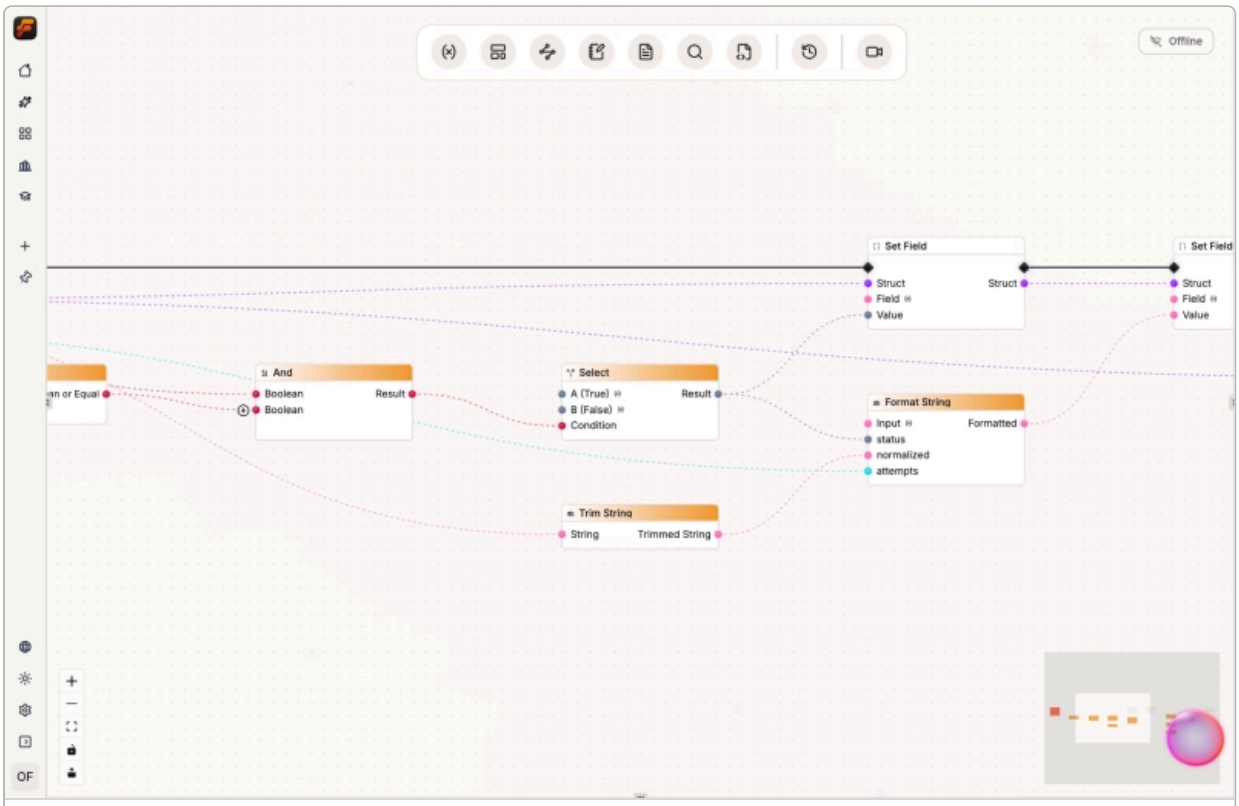
That form needs a typed numeric field. An open Struct field is Generic until the Flow proves more, so an author may have to read and convert it explicitly. `any` does not become safe merely because the uncertainty sits behind a dot.

9.3 Conditional expressions select values

A ternary is a value decision:

```
let status = urgent ? "critical" : "investigate"
```

It lowers to the Types Select node:



The ternary becomes one Select data node. Its result can feed several consumers without creating execution branches.

This is not an execution branch. At runtime the Select node evaluates its condition and reads only the selected data input, but it exposes no alternative Execution arms. An impure operation placed inside an expression can also need its own place in the surrounding execution chain, so a ternary must not be used as a substitute for conditional side effects. Use a ternary for value-level selection. Use `if` or a named execution-arm block when only one side should call an external system, write data, or handle a failure. Chapter 10 develops that distinction through execution paths.

The condition must resolve to a Boolean input, and the two alternatives must be usable by the same selected output contract. With ordinary literals that is straightforward. With distinct schemas or container shapes, an explicit common boundary is safer than erasing the disagreement through Generic.

The renderer recognizes the current `utils_types_select` node specifically. That node presently has only `condition`, `a`, and `b`, so the ternary reproduces its full data contract. Unlike binary operator lowering, this path

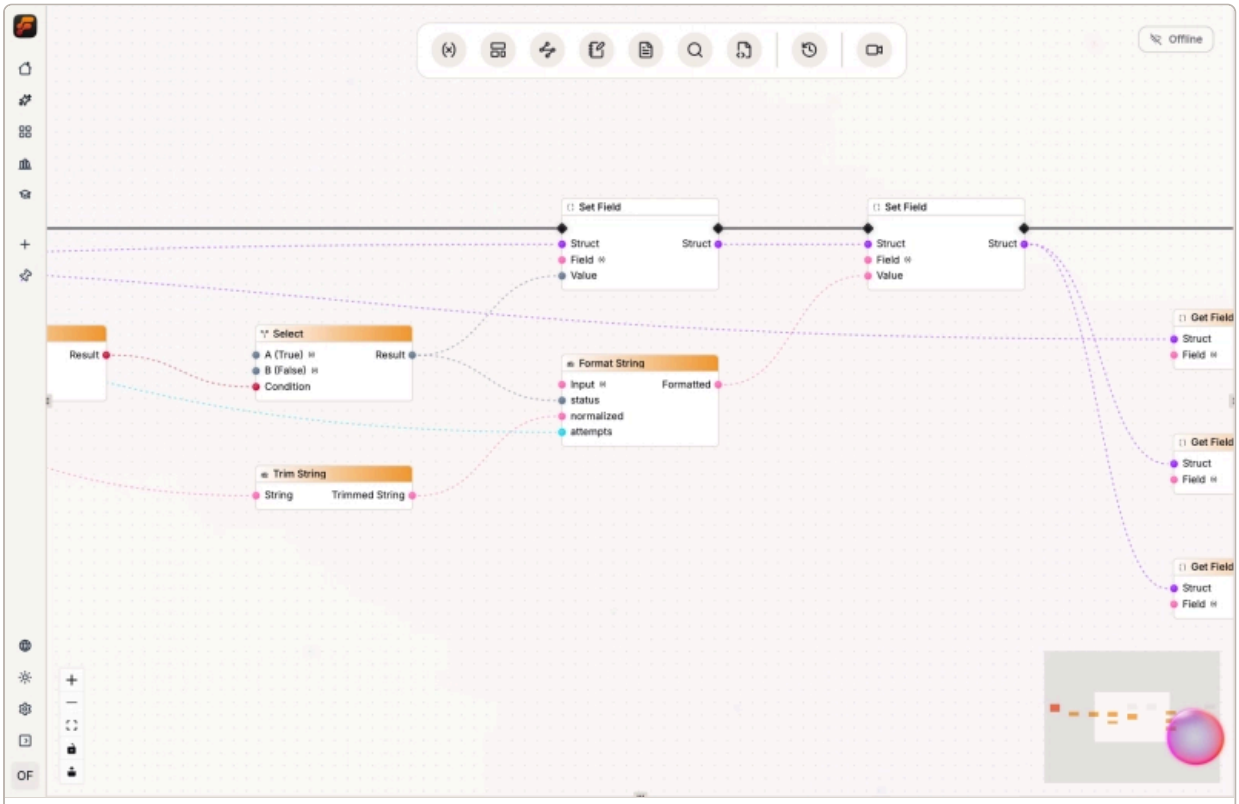
does not yet guard against future extra configured inputs. If the Select node grows another meaningful pin, the lossless-sugar test must grow with it.

9.4 Template literals are String Format nodes

Template literals turn a format operation into readable prose:

```
let summary = `${status}: ${normalized} after ${attempts} attempt(s)`
```

Apply converts that expression into one String Format node. Static text becomes its `format_string` input. Each interpolation becomes a dynamic input pin:



Template interpolation names survive as dynamic Studio pins, so every placeholder has an inspectable producer.

Simple references keep their names. A member or output access normally uses its final segment. More complex expressions receive stable positional names such as `arg1`. If two different expressions want the same place-

holder name, the later one receives a suffix. Repeating the same bare reference reuses one placeholder pin, just as one graph output may fan out to several places in a format string.

This naming is not cosmetic. Dynamic pins are part of the node. The Board needs to know which wire supplies `{status}`, and text reconstructed from the Board must be able to produce the same set of pins again.

Template text therefore has one deliberate edge. Literal text shaped exactly like a format placeholder, such as `{status}`, cannot be treated as inert braces, because the underlying String Format node would interpret it as a pin. FlowScript rejects that ambiguous template. If a placeholder was intended, write the explicit format call and supply it; if literal braces were intended, change the text rather than relying on an escape the node itself cannot preserve.

The renderer returns to template syntax only when the round trip is exact. It requires:

- a literal format string;
- a value or wire for every placeholder;
- no extra meaningful input pins; and
- placeholder names that would be regenerated identically from the expressions.

If one of those checks fails, the Board renders the explicit call:

```
string::format({
    formatString: "Incident {ticket}",
    ticket: externalId,
})
```

That is the correct fallback. A slightly longer line is preferable to a pretty template that silently renames a dynamic pin or drops a configured value.

9.5 Field reads and writes are value flow

A dot can select two different graph concepts:

```
let found = split.found
let status = incident.status
```

The first expression may name an output pin on the node bound to `split`. The second reads a field from a Struct. Reconciliation tries the declared node output first. If no such output exists and the base is Struct-shaped, it lowers the expression to the appropriate Struct field-access node. A named interface lets that access retain the field's declared type; an open Struct produces a more generic boundary.

There is a current read-side round-trip bug to keep separate from that intended rule. Generic Get Field exposes both `value` and `found`. Board-to-text lowering currently recognizes the node as a member access without first checking which output is wired. A wire from `found` can therefore render as `incident.status`, which ordinarily means the value. The safe authoring form when the presence flag matters is an explicit call and named output until lowering checks the selected pin.

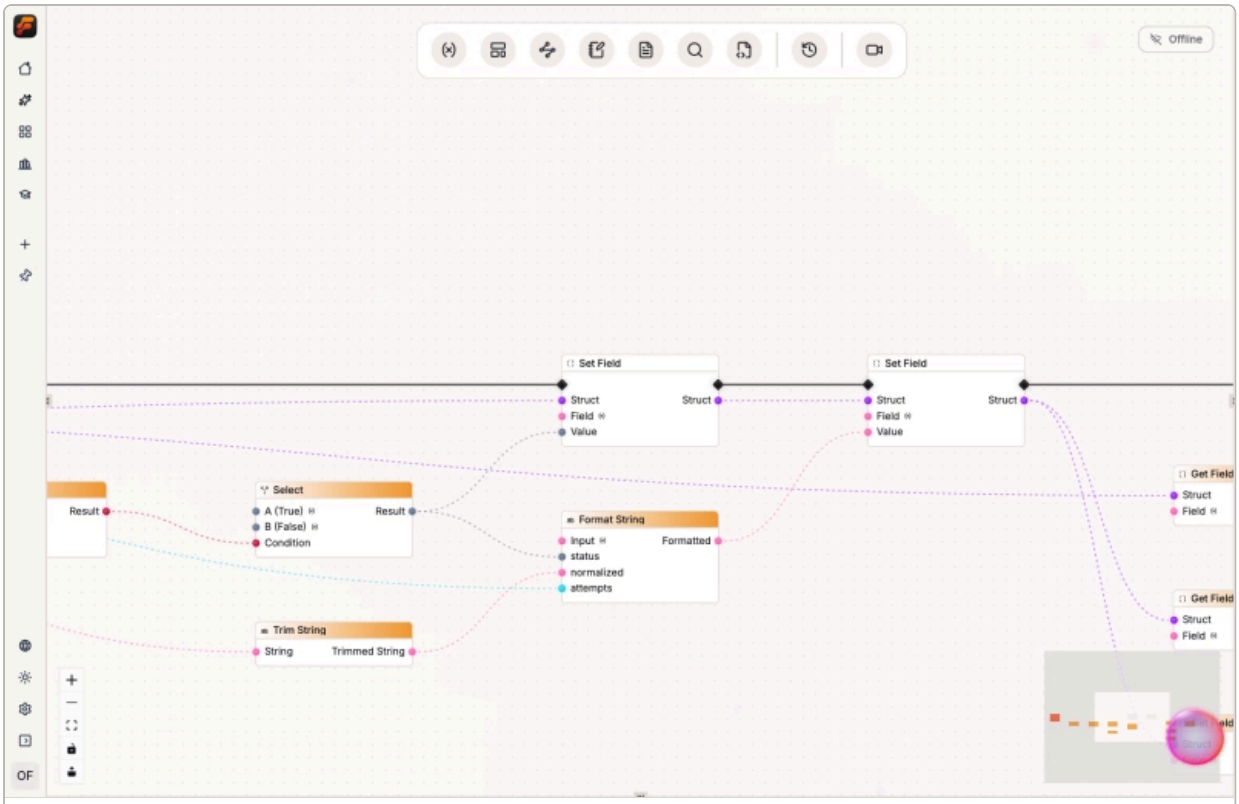
```
const { value: status, found } = incident.get({ field: "status" })
```

A field write is equally concrete:

```
incident.status = "closed"
```

It lowers to Set Field with three data inputs—`struct_in`, the literal path `"status"`, and the new value—and a `struct_out` output. FlowScript then rebinds `incident` so later uses resolve to that output.

It helps to see the successive values on the Board:



Each Set Field produces a new Struct wire for the next update; existing consumers remain connected to the earlier producer.

This is the precise sense in which the source appears to mutate a Struct. A consumer wired before the assignment keeps `incident@0`. A reference after the first write receives `incident@1`; after the second, it receives `incident@2`. The pins on each operation carry that operation's current runtime value. The source name is the readable binding that selects which producer subsequent expressions use.

If the name is never used after a field write, the updated Struct is simply an unused copy. If it is used, it is the new version. Nothing travels backward through an existing wire to change a consumer that already received the earlier producer.

Nested literal paths follow the same model:

```
incident.owner.name = "Ada"
incident.affectedSystems[0].status = "degraded"
```

When the field itself is computed rather than a literal path, the compact dot assignment cannot name it honestly. The graph keeps an explicit Set Field call instead. The renderer also uses dot assignment only for an accu-

mulator-shaped chain where the Set Field output becomes the next value of the same binding. A seed operation, a cross-source update, or a wired dynamic field remains an explicit call.

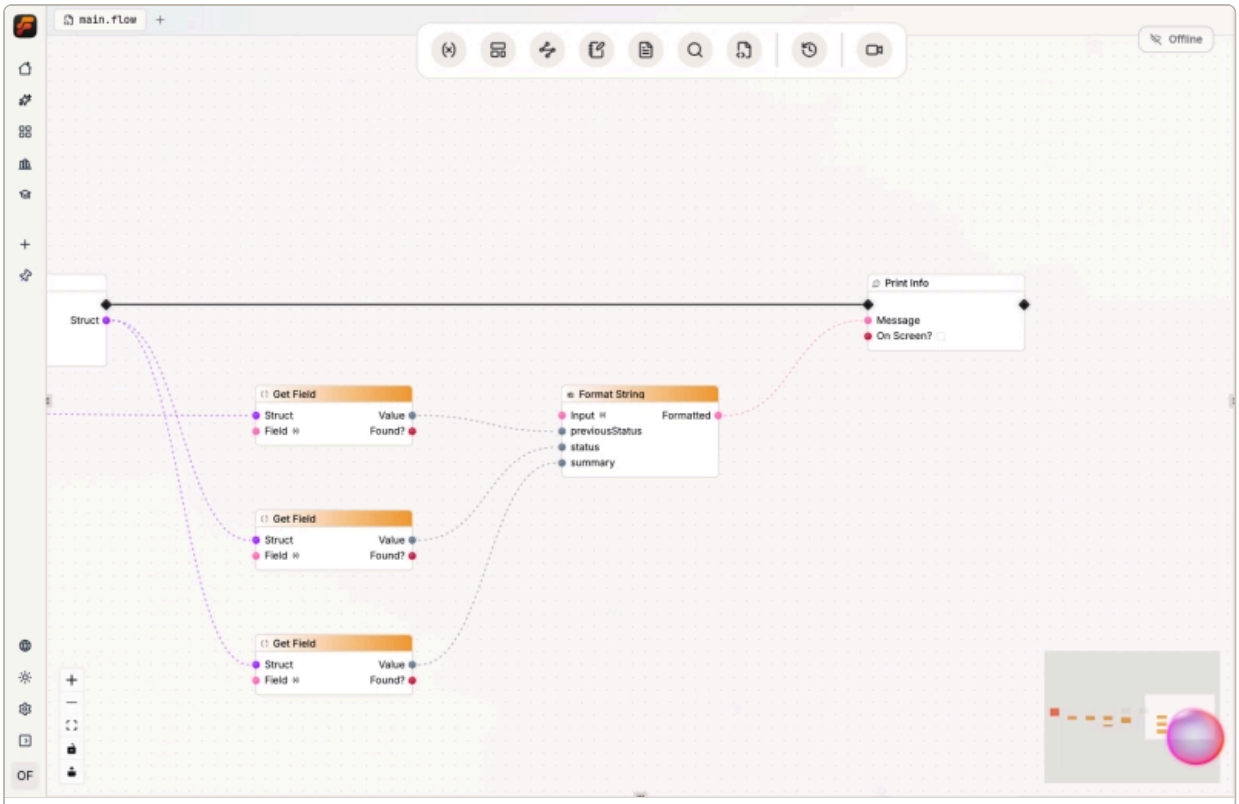
9.6 Sugar must round-trip honestly

The chapter's complete example applies all four kinds of readable sugar to the Incident Triage Flow:

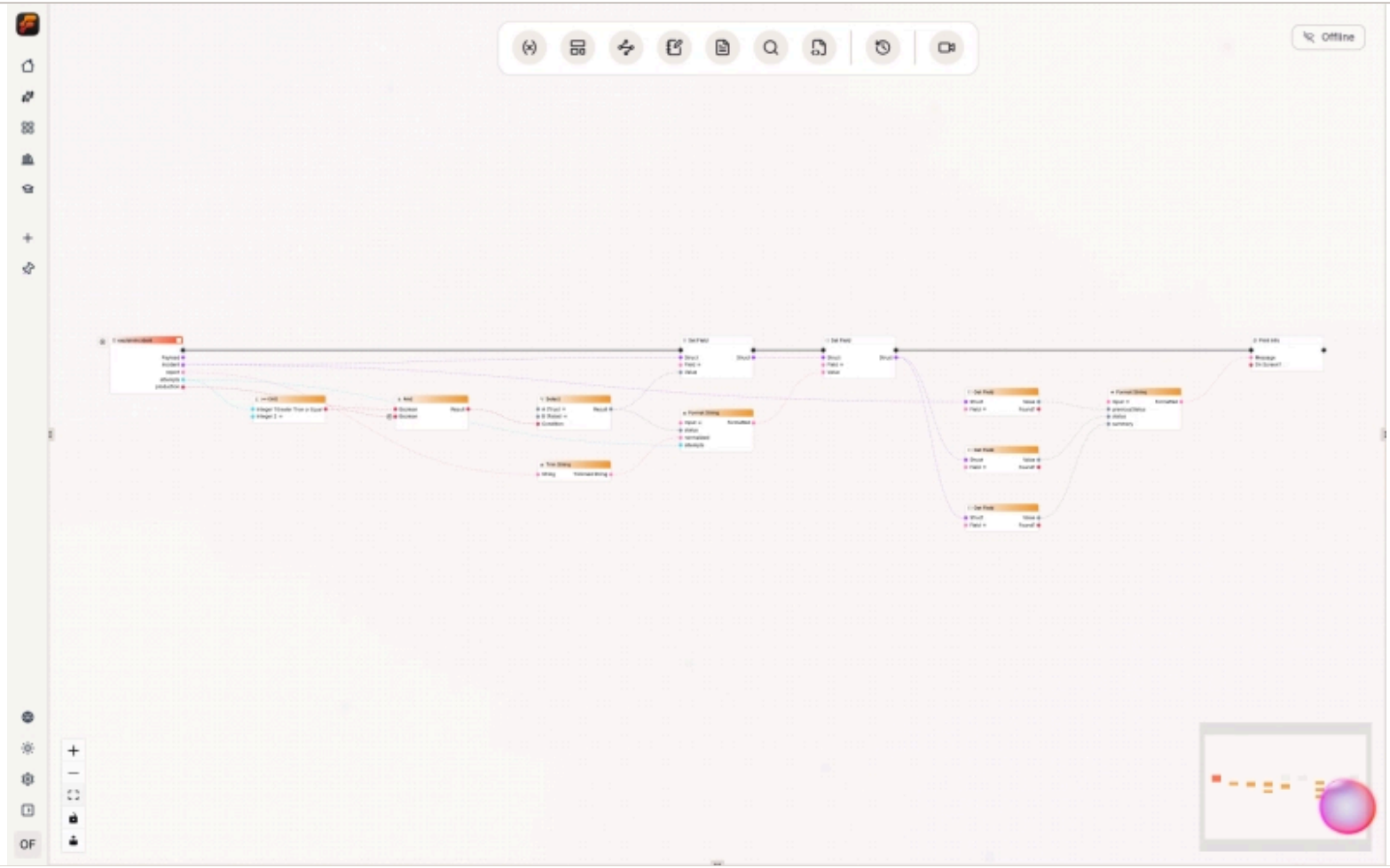
```
use log::{ info }

eventsGeneric explainIncident(payload: Struct, incident: Struct, report: string, attempts: int, production: bool) {
    const normalized = report.trim()
    let urgent = production && (attempts ⩾ 3)
    let status = urgent ? "critical" : "investigate"
    let previousStatus = incident.status
    incident.status = status
    incident.summary = `${status}: ${normalized} after ${attempts} attempt(s)`
    info({ message: `${previousStatus} → ${incident.status}: ${incident.summary}`, toast: false })
}
```

Read it once as code. It trims a report, classifies repeated production trouble, remembers the old status, produces two successive Incident values, and logs the transition. Then read the same logic as a graph:



The final focused region completes the graph detail shown in the three earlier crops: previous and updated Struct fields converge into the last format and log call.



The complete readable-sugar example is an overview at page width; the focused renders expose its details. Open this image for the full-size graph.

The exact layout is Studio's concern, but every meaningful operation in the source has a node or a pin value behind it. Conversely, every meaningful configured input on those nodes is intended to survive when the Board becomes text. The release gaps above identify where current lowering still falls short of that contract.

That second direction is where lossless sugar earns its name. Current lowering checks more than a node type:

SHORT FORM	IT IS RENDERED ONLY WHEN...	OTHERWISE...
`a op b`	the node is a supported operator shape, its two operands are identifiable, and any omitted trailing inputs are untouched defaults	keep the catalog call
`c ? a : b`	the node is the recognized Types Select contract; today it has exactly the three represented inputs	keep the catalog call
`` `…${x}…` ``	the literal format and complete placeholder-pin set regenerate exactly	keep `string::format(...)`
`record.field = value`	a literal-path Set Field is rebinding the same Struct accumulator	keep `struct::set(...)`

String equality makes the rule tangible. With the case option untouched, a graph can render:

```
let same = left == right
```

When the node's `ignoreCase` input is enabled, the operator can no longer carry the contract, so the renderer preserves the setting:

```
const same = left.equal({ string: right, ignoreCase: true })
```

Applying or switching views may therefore make source longer. That is not a round-trip defect. It is the language refusing to lie about the graph.

Current Struct accumulator rendering makes that concrete in the chapter example. The first Set Field may read back as an explicit seed alias—possibly with a generated name such as `record`—and only the following same-accumulator update may return to `record.summary = …`. The parser fix-

ture above is in canonical source form, but it is not a promise of byte-identical Board readback. The contract is the preserved graph and its configuration.

The complementary failure is equally useful. Change the example to:

```
let urgent = report + attempts
```

The parser can recognize the expression, but Apply cannot select one correct node family. It reports the String/Integer contradiction at that expression and does not manufacture a conversion. The repair is local and visible: choose formatting if this is a label, or choose a typed parse and handle its success if this is arithmetic.

These constraints are what make familiar syntax safe in a dual-view language. Operators are typed catalog operations. Ternaries select values rather than disguising execution branches. Templates retain their dynamic pins. Field assignment advances a visible Struct value instead of mutating the past. Whenever the compact notation cannot carry the entire decision, FlowScript shows the node call.

The next chapter adds visible execution structure: `if`, named outcome arms, sequential and parallel loops, `while`, and `return`. The same standard will apply there—short syntax is welcome only when every path still has an honest place on the Board.

PART II · CHAPTER 10

Branches, Loops, Parallelism, and Return

Build visible control flow with if branches, named execution arms, sequential and parallel loops, bounded while, stopping, and return boundaries.

Control flow in FlowScript looks familiar:

```
if (critical) {
    notifyOperations()
} else {
    recordForReview()
}

for (const report of reports) {
    classify(report)
}
```

But it does not disappear into an invisible instruction pointer. An `if` is a node with visible execution outputs. A collection loop owns an iterable, exposes the current value and index, and repeatedly triggers its body path. While owns a condition and maximum iteration count. The statement after either loop is connected to **Done**. A `return` either wires data to a function boundary or terminates one Event execution branch.

This leads to the central rule of the chapter:

> *Control flow is topology. Every path that can run, fail, repeat, or finish should remain visible in both source and the workflow.*

That visibility matters most when something goes wrong. During the 3 a.m. incident from Chapter 1, the only initial fact was a domain report: production was on hold. A Flow-Like implementation could not guarantee that the underlying dependency would never fail. It could make the decision, the attempted call, its expected Error route, any unexpected failure, and the recovery work inspectable at the responsible nodes.

Release check: The current language supports Boolean branches, named execution arms, sequential and parallel collection loops, bounded `while`, and return expressions. Several details deliberately differ from TypeScript. `@parallel` currently implies a concurrency limit of 30. Plain `while` implies at most 15 iterations. FlowScript has no `break` or `continue` statement. Function `return` does not yet perform function-wide early termination. The current sequential, parallel, while, and break-capable loop nodes also log and absorb errors returned by their child paths. They do not fail fast solely because a child returned an error: collection loops continue remaining items, While reevaluates its condition, and the break-capable node then samples Break. The enclosing run can still finish with a Success status despite Error evidence. That conflicts with Flow-Like's intended aggregate-failure invariant and is an implementation gap, not a reliability guarantee.

10.1 `if`, `else if`, and `else`

Use `if` when a Boolean value chooses an execution path:

```
if (report.contains({
    substring: "production is on hold",
    ignoreCase: true,
})) {
    error({ message: report, toast: false })
} else {
    info({ message: report, toast: false })
}
```

The condition resolves to a Boolean data input on a branch node. The two blocks connect to its True and False execution outputs. At runtime one of those outputs becomes active. Moving one block below another on the canvas cannot change the choice; only the wires and the condition do.

This is different from the ternary in Chapter 9:

```
let status = critical ? "critical" : "investigate"
```

The ternary selects one value. `if` selects an execution path. Use `if` when the alternatives contain calls, writes, retries, tickets, or other work that should run only on the selected side.

FlowScript accepts an `else if` ladder:

```
if (productionStopped) {
    status = "critical"
} else if (degraded) {
    status = "warning"
} else {
    status = "healthy"
}
```

Canonical source currently renders the same graph as a nested branch:

```
if (productionStopped) {
    status = "critical"
} else {
    if (degraded) {
        status = "warning"
    } else {
        status = "healthy"
    }
}
```

The second form shows the actual topology: the second decision is reached only through the False arm of the first. The indentation is not merely style; it describes which branch owns the next decision.

A branch is not an error handler

A False arm means that a Boolean condition evaluated to false. It does not catch an error thrown while calculating the condition or running the True arm. Expected negative outcomes should have their own modeled execution output. Unexpected node failures use the node's **Handle Errors** path when the Flow is designed to recover from them. Those are separate contracts, which is why the visual workflow can show them separately.

10.2 Named execution arms

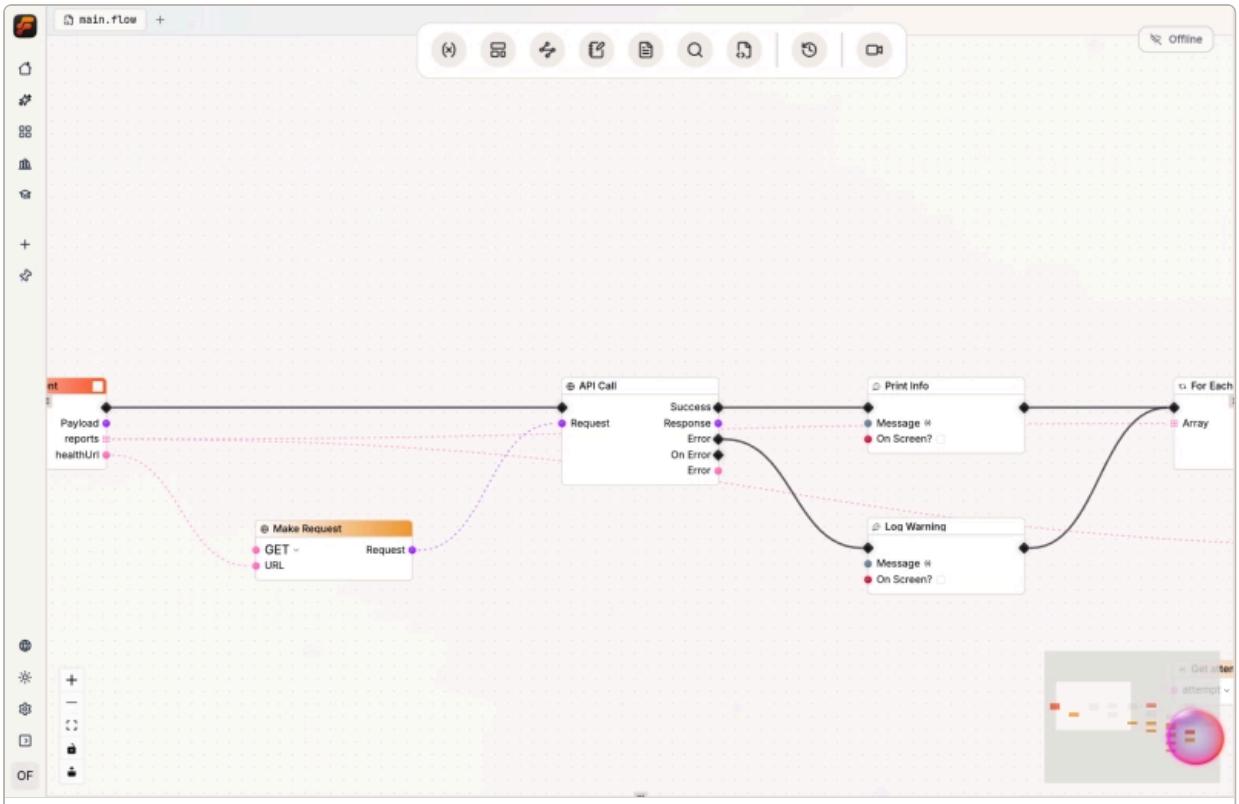
True and False are not sufficient names for every decision. The built-in HTTP API Call node, for example, has Success and Error execution outputs. FlowScript preserves those labels with a named arm block:

```
const request = http::request({ method: "GET", url: healthUrl })
const apiCall = http::fetch({ request: request })
apiCall {
    execSuccess: {
        info({ message: "Dependency is reachable", toast: false })
    }
    execError: {
        createIncidentTicket()
    }
}
```

`apiCall` is a handle to the call. It can expose data outputs such as `apiCall.response`, while the block below it connects work to the call's execution pins. A node with three or more outcomes uses the same shape with one named block per connected arm. This is clearer than pretending every multi-outcome operation is a Boolean or hiding its routes behind exceptions.

For the current HTTP node, a completed 2xx response selects Success. A completed non-2xx response selects Error. A transport failure, an invalid request value, or a failure while reading the response can instead be an unexpected node error. Enabling **Handle Errors** adds the generic `On Error` execution path and an Error string output for that second category.

The distinction is operationally useful:



The source-authored Success and Error arms remain wired; this capture applies Studio's Handle Errors toggle so the generic On Error execution and Error string outputs are visible.

A Flow can route Error to a retry, ticket, or domain response without claiming that the node crashed. If an On Error recovery path completes, the runtime can continue while the original node still carries failed evidence. Handling an error does not erase where it happened.

Arm names are part of the node contract. Do not rename `execError` to something friendlier in source and assume it means the same thing. Canonical source camel-cases the internal pin name, such as `exec_error` to `execError`; Apply can also recognize supported raw and friendly spellings. Use the Board's canonical rendering, editor completion, or Apply's available-output diagnostic instead of guessing.

Some two-output nodes render in compact `if (nodeCall())` form with comments such as `// exec_out_exists` and `// exec_out_missing` after the braces. Those comments are semantic pin labels, not disposable prose; they preserve which graph output each arm represents.

10.3 Sequential `for … of`

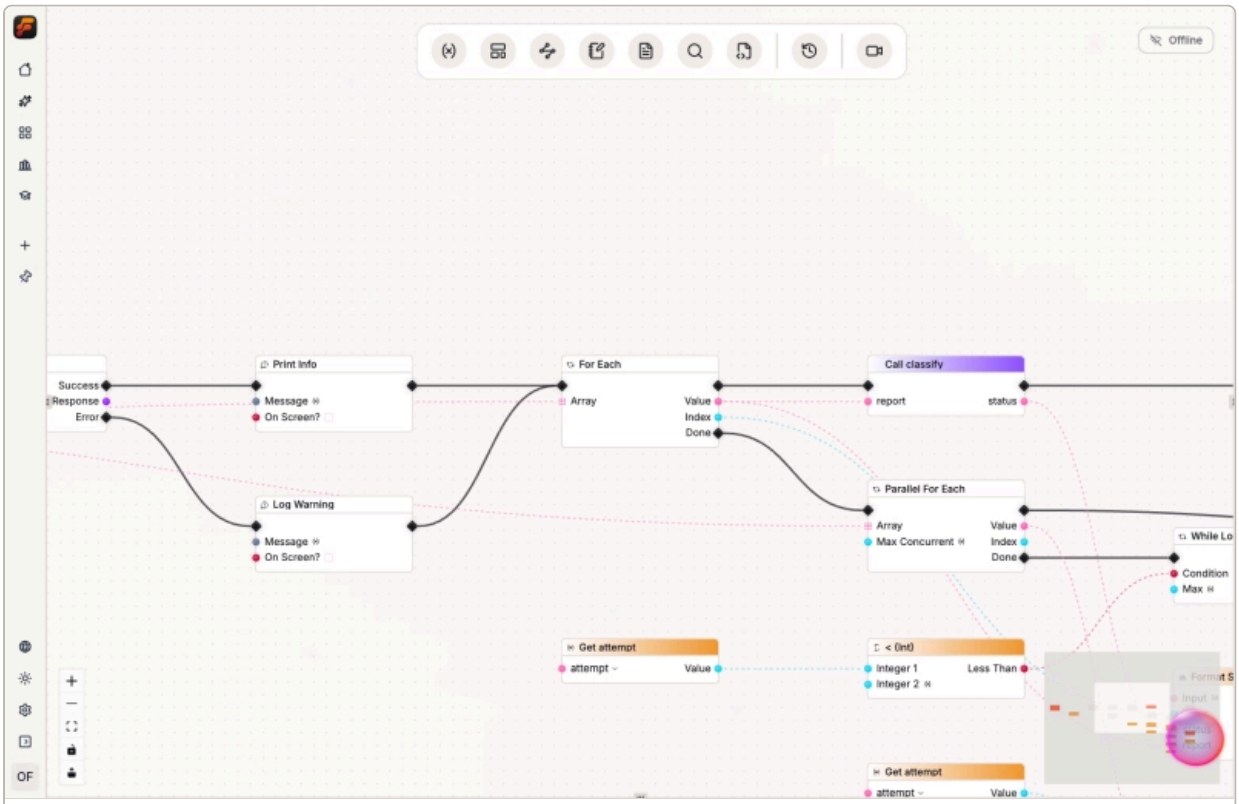
The ordinary collection loop is sequential:

```
for (const report of reports) {
    classify(report)
}
```

Bind the zero-based index as well when identity or ordering matters:

```
for (const [index, report] of reports) {
    info({ message: `#${index}: ${report}`, toast: false })
}
```

The loop evaluates its array once, then processes values in input order. For each element, it publishes the current value and index and waits for the connected body chain to settle before starting the next item. Only after every item does the **Done** output activate and allow the statement following the loop to run.



The static Board shows one body path: the runtime reuses it for each array item in order, then activates Done once.

This makes ordinary `for` the safe default when iterations touch shared or external state:

- an API with a strict rate limit;
- a ticket system where order or deduplication matters;

- a database update whose later write depends on an earlier one;
- a model call whose concurrent spend or provider rate must remain bounded; or
- any operation whose idempotency is not yet proven.

Sequential does not mean fail-fast today. The For Each node owns the iterations. If a body chain returns an unhandled error, that chain stops, the loop records an Error message attributed to the loop node and naming the iteration, and the next item still runs. Logs emitted inside the failed chain retain attribution to their child nodes. A modeled or handled failure can likewise route to ticket creation and then allow later items to proceed. This behavior is useful for batch-style work in which one bad record should not discard all good records.

It also creates a current status caveat. The loop node itself returns success after logging a child error, so the enclosing run may report Success even though a child trace and Error log show a failure. Flow-Like's intended invariant is stricter: any unhandled child failure should make the aggregate run Failed after the remaining work settles. Until that invariant is uniformly implemented, a production Flow that must report "all records succeeded" should model that fact explicitly—capture and collect per-item outcomes, then make the final success decision visible.

10.4 `@parallel for`

Parallelism is an opt-in promise that iterations may overlap:

```
@parallel
for (const report of reports) {
    inspectIndependentReport(report)
}
```

The current Parallel For Each node starts work for independent items up to its configured concurrency limit. The `@parallel` sugar is lossless only at the node's default of 30. A custom limit remains an explicit call so the meaningful setting cannot disappear:

```
for (const parallelForEach of control::parallelForEach({
    array: reports,
    maxConcurrent: 5,
})) {
    inspectIndependentReport(parallelForEach.value)
}
```

`maxConcurrent: 1` schedules one task at a time. A positive value bounds active item/body-root tasks; most loops have one body root, so it behaves as the familiar number of active iterations. `0` is documented as unlimited, and the current implementation treats any non-positive value that way. Unlimited concurrency is rarely a responsible setting for an external service.

Done is a barrier, not an ordering guarantee

Parallel For Each waits for every child to settle before it activates Done. It does not expose a collected-results array, and side effects may complete in any order. Carry the original index, collect the per-item results, and sort them when order matters. The catalog's execution-only Gather node can provide a barrier, but it does not collect data values for you. Never infer result order from the vertical position of nodes, log arrival time, or the order in which parallel branches appear to finish.

A failing child does not cancel its siblings. The current node continues scheduling remaining items, drains all child work, logs each returned failure, activates Done, and returns success from the loop node itself. The aggregate-status gap described for sequential loops therefore also applies here.

Parallel execution is not automatically faster or cheaper. Before opting in, answer these questions:

1. Can the iterations safely run in any order?
2. Are writes idempotent, or can a retry create duplicates?
3. What are the downstream rate and connection limits?
4. What is the maximum acceptable simultaneous model, network, memory, and cost load?
5. How are individual failures represented and collected?
6. What should happen to already-running siblings after one failure?

The production default is simple: keep external API calls, model calls, ticket creation, and database writes sequential until bounded concurrency has been justified. Parallelize independent work because its contract permits overlap, not because an annotation is available.

10.5 Bounded `while`

Workflow loops must have an operational ceiling. The compact form is:

```
let attempt = 0
while (attempt < 3) {
    pollDependency()
    attempt = attempt + 1
}
```

Before each body run, While retriggers the dependencies of its condition and evaluates the Boolean again. The body runs only while that value is true. The current node also has a `maxIter` guard. A plain `while (condition)` preserves the default value of 15; changing the guard makes the node call explicit:

```
while (control::whileLoop({ condition: retryReady, maxIter: 3 })) {
    pollDependency()
}
```

The maximum bounds the number of body starts and prevents an accidentally true condition from starting bodies forever. It does not prove that the loop made progress, enforce a wall-clock deadline, add delay or backoff, or cancel a slow operation inside the body. Those policies still belong to the Flow and the called nodes.

There is another current edge to understand: if the condition remains true at the maximum, While silently activates Done. It has no separate Exhausted output and emits no warning merely because the ceiling was reached. A Flow in which exhaustion means failure must make that state explicit, for example by maintaining an attempt value and checking it after the loop before choosing Success, Retry Later, or Escalate.

Body failures currently follow the same ownership rule as collection loops: they are logged, the loop proceeds to reevaluate its condition, and it can still finish successfully. A failure while directly evaluating the condition can propagate, while a failure to retrigger a dependency is currently logged before the loop exits through Done. These details are release-sensitive and deserve runtime tests before a Flow relies on them.

Cancellation is cooperative. A stopped run is observed at node boundaries, and long-running nodes must cooperate with cancellation themselves. The plain loop nodes do not currently stop walking or scheduling iterations as consistently as the batch-loop variants. A maximum iteration count and timeouts on external operations therefore remain useful even when a caller can cancel the run.

10.6 Stopping and skipping are structural today

FlowScript does not currently have `break` or `continue` statements. To skip the remainder of one ordinary iteration, put that remainder behind a branch:

```
for (const report of reports) {
    const relevant = isRelevant(report)
    if (relevant) {
        classify(report)
        persistFinding(report)
    }
}
```

The False arm contains no further work, so that execution path reaches the end of the body and the loop advances. On the Board, the skipped path is obvious rather than hidden in a jump statement.

The catalog also contains a separate **For Each (Break)** node. It samples a Boolean Break input before the loop and after each body root; a true value stops any remaining body roots and later items, then activates Completed. That node is available visually but is not currently part of FlowScript's structured loop registry, so it does not round-trip as a `break` keyword or ordinary `for` sugar. Treat it as an explicit catalog capability and verify its rendered source against the target release.

This is a real authoring gap, not an invitation to emulate arbitrary jumps. A future text form should preserve the same visible stop condition and Completed path in both views.

10.7 `return` is a boundary, not stack unwinding

In a function, return values map positionally to the declared output pins:

```
function classify(report: string): (status: string, normalized: string) {
    const normalized = report.trim()
    let status = "investigate"
    if (normalized.contains({ substring: "production is on hold", ignoreCase: true })) {
        status = "critical"
    }
    return status, normalized
}
```

The first expression wires to `status`; the second wires to `normalized`. Literals can be materialized as typed values, and a previously bound call output can supply a return pin. Validation reports extra return values or declared output pins left without a source rather than guessing how an arity mismatch should be repaired.

What this does **not** mean today is JavaScript-style early return:

```
// Do not rely on this shape to terminate the whole function today.
if (invalid) {
    return "rejected"
}
performSideEffect()
```

The current function reconciliation wires data to layer output pins; there is not yet a function-wide execution-terminating Return node. FlowPilot's intermediate representation consequently rejects nested function returns and permits a single final, unconditional top-level return. Write functions in that supported shape. When early selection is needed, branch to compute or assign the result, rejoin, and return it once at the boundary.

An Event return has a different implementation:

```
eventsGeneric status(payload: Struct) {
    return "accepted"
}
```

It becomes a terminal Return Result node with no execution successor. It ends that execution branch and publishes one Event result. An Event accepts at most one return value; wrap several fields in a Struct when the caller needs a compound response.

Even an Event return does not cancel sibling execution branches. In addition, current execution surfaces do not all aggregate several competing result events the same way. Synchronous server and remote callers usually retain the first emitted result. Subcontext merging overwrites with the last merged child result, UI run state displays the last result event it received, and streaming SSE forwards every result event. Parallel timing makes any implicit choice a poor business contract.

The robust rule is therefore:

> *Produce one logical result path per invocation. Do not race several* `return` *statements and let timing choose the caller's answer.*

When branches can finish with different domain outcomes, join them through explicit data and make one final selection before the result boundary. That design is easier to read, test, trace, and eventually migrate to a true function-wide Return node.

10.8 Read the whole control-flow example

The Chapter 10 fixture combines the forms in one small incident coordinator:

```
use log::{ info, warn }

function classify(report: string): (status: string) {
    let status = "investigate"
    if (report.contains({ substring: "production is on hold", ignoreCase: true })) {
        status = "critical"
    }
    return status
}

eventsGeneric coordinateIncident(payload: Struct, reports: string[], healthUrl: string) {
    const request = http::request({ method: "GET", url: healthUrl })
    const apiCall = http::fetch({ request: request })
    apiCall {
        execSuccess: {
            info({ message: "Dependency is reachable", toast: false })
        }
        execError: {
            warn({ message: "Dependency returned an unsuccessful response", toast: false })
        }
    }
    for (const [index, report] of reports) {
        const status = classify(report)
        info({ message: `#${index} ${status}: ${report}`, toast: false })
    }
    return "incident coordination completed"
}
```

Read the same source as a workflow. The Event opens an execution path. API Call splits that path into named outcomes. Because both arms contain continuing work, the statement after the block fans in from both arm tails; the HTTP node itself has no separate Done output. For Each owns the reports, exposes value and index, and joins at Done. The function exposes one output pin. Return Result terminates the Event branch and supplies the caller's value.

The complete canonical fixture also includes `@parallel for` and bounded `while`, and is checked by the repository's FlowScript parser/renderer test. It remains deliberately small enough to inspect in both views. Complexity belongs in visible layers and functions, not in control flow that only one author can mentally simulate.

The goal is not to make FlowScript look like TypeScript at any cost. The goal is to make familiar control structures honest projections of a workflow whose paths, limits, failures, and results a system expert can understand during the next 3 a.m. call.

PART II · CHAPTER 11

State, Configuration, Runtime Values, and Secrets

Choose the right scope and trust boundary for local bindings, Flow variables, runtime configuration, BYOK secrets, cache data, files, and durable state.

An API address, a retry limit, an OpenAI key, the time of the last successful synchronization, and a ten-gigabyte reference dataset are all “values.” They are not the same kind of state.

Putting all five into global variables would be convenient for a few minutes and expensive for the rest of the application’s life. Their owners, lifetimes, sizes, readers, and failure behavior are different:

VALUE	APPROPRIATE HOME IN THE INCIDENT APP
API base URL	Exposed Flow configuration
Retry limit	Exposed Flow configuration
Preferred language	Exposed Flow configuration
User’s OpenAI API key	Secret runtime configuration for local BYOK
Last successful synchronization	Cache if it is only a hint; database if it is a durable fact
Reference data	File, cache, or database according to size, freshness, and query needs

The syntax is the easy part. The design question is: **who owns this value, and how long must it remain true?**

> *State should live at the narrowest scope that satisfies its real lifetime. A Flow variable is not a tiny database, a cache is not a system of record, and source code is never a credential store.*

Release check: Each run currently receives fresh mutable Flow variables. Top-level `const` means non-exposed and top-level `let` means exposed; neither means immutable. `@runtime` and `@secret` can be combined. Web and Desktop store configured values in local IndexedDB, keyed by App and variable, so the defensible current wording is **device-profile local**, not explicitly user-keyed. Interactive execution paths prompt for missing values before running, but direct or unattended paths do not uniformly reject them: the runtime can still fall back to a Board/Event default or `null`. `@readonly` currently disables editing surfaces but does not block Set Variable during execution. Per-user/device identity, universal configuration preflight, and runtime enforcement of `@readonly` are intended contracts that are not complete today.

11.1 Local bindings and Flow variables

Inside a function or Event body, a binding usually names a value produced by the graph:

```
const normalized = report.trim()
const urgent = normalized.contains({
    substring: "production is on hold",
    ignoreCase: true,
})
```

`normalized` is not an independent storage slot. It names the output of String Trim. `urgent` names the Boolean output of String Contains. In the workflow, consumers connect directly to those producer pins.

A mutable local can model an accumulator:

```
let attempts = 0
attempts = attempts + 1
```

The reconciler can materialize a local variable and its Get/Set nodes when execution paths need a shared mutable value. Its lifetime is still the current run. Starting the Event again creates new in-memory state from the configured defaults.

Top-level declarations describe Board variables:

```
const urgentPhrase = "production is on hold"
let retryLimit = 3
```

Nodes throughout the Flow can read or write them through generated Get and Set operations. Every run receives its own variable map, so two concurrent invocations do not deliberately communicate through `retryLimit`. A write in one run does not update the declaration for the next run.

This makes Flow variables suitable for:

- values shared by several nodes during one invocation;
- counters and flags used by control flow;
- arrays or Structs accumulated during a run; and
- configured defaults that the Flow may temporarily transform.

They are not suitable for a last-sync record, account balance, inventory count, durable job state, or anything expected to survive a new invocation. Put those facts in App Storage or a database.

11.2 Top-level `const` and `let` describe exposure

TypeScript readers need to unlearn one association at document scope:

DECLARATION	CURRENT FLOWSCRIPT MEANING
`const value = …`	Create a non-exposed Board variable
`let value = …`	Create an exposed Board variable

Both runtime values are mutable today. The difference is configuration visibility.

An exposed and editable `let` appears in the App's Configuration screen. A person authorized to change the App or Board can adjust the value without opening FlowScript or rearranging a workflow. Events can also carry permitted overrides for exposed variables. The exact people who may do that are decided by the App's permission policy, not by `let` itself.

For the Incident App, these are reasonable exposed defaults:

```
@description("Base URL used by the incident AI provider")
@category("Incident AI")
let apiBaseUrl = "https://api.openai.com/v1"

@description("Maximum number of provider attempts")
@category("Incident AI")
let retryLimit = 3

@description("Language used for generated incident explanations")
@category("Incident AI")
let preferredLanguage = "en"
```

The Flow author decides which values are configuration at all. App configurators decide their shared defaults later. A caller should not be allowed to rewrite an arbitrary non-exposed variable merely because it can guess its ID.

Changing an exposed default changes Board configuration; it is not a private preference for just one runner. If language truly belongs to each person or device, make it runtime-configured instead. If production and development need different values for the same unattended Event, configure those Event or deployment boundaries explicitly rather than making the Flow detect its environment through hidden assumptions.

11.3 Decorators carry platform consequences

Variable decorators are metadata that both authoring views can preserve:

DECORATOR	MEANING TODAY
`@description("…")`	Explain what the configurator must provide
`@category("…")`	Group related values in configuration interfaces
`@readonly`	Mark the variable non-editable in current authoring/configuration UI
`@runtime`	Take the configured value from the runtime channel
`@secret`	Hide the value from FlowScript and sensitive authoring/read paths
`@schema("…")`	Attach legacy inline Struct schema metadata

Prefer a named interface over `@schema` when source should describe a Struct contract. Decorators must sit immediately above their declaration. Canonical order is description, category, legacy schema, secret, readonly, then runtime.

`@readonly` is currently weaker than its name

This declaration describes a value that App configuration should not edit:

```
@description("Provider selected by the Flow author")
@readonly
const providerName = "OpenAI"
```

Today, `@readonly` sets the Board variable's `editable` flag to false. The variables panel and configuration UI disable edits, but a Set Variable node can still change the in-memory value during a run. That is an implementation gap. The intended meaning should become a real runtime write barrier as well; until then, do not treat `@readonly` as a security boundary or integrity guarantee.

This is deliberately separate from top-level `const`. `const` controls exposure. `@readonly` expresses whether mutation is allowed. Those concepts deserve separate syntax because a hidden value can still be mutable, while an exposed reference value may eventually need to be visible but immutable.

11.4 Runtime configuration is runner-specific input

Use `@runtime` when the Flow definition should contain the contract but not one shared value:

```
@description("Path to the local incident export directory")
@runtime
const incidentExportPath: Path
```

The Flow retains the name, type, description, and decorators. The standard Web and Desktop clients store the configured JSON value outside the Flow in local IndexedDB. A saved value overrides the Board default for that run. A non-secret runtime value may be included in a remote invocation payload.

The intended scope is per user and device. The current store does not yet encode both dimensions: its record key is the App ID plus variable ID, inside that browser/Desktop profile. This gives device-profile isolation, but it does not independently distinguish two Flow-Like accounts using the same local profile. The implementation and documentation should be aligned before the book promises a strict user-and-device key.

Interactive execution currently follows a useful preflight:

```
Run requested
    │
    ├─ saved records present ─▶ execute
    │
    └─ values missing ─▶ configuration dialog ─▶ save ─▶ execute
```

The founder's rule is stronger: a required runtime value must be configured before any node runs. That should hold for interactive, API, scheduled, internal, and agent-initiated execution. Currently the interactive execution service performs the check, but that check proves only that a record was saved—not that its value is still valid for the consuming node. The core runtime does not enforce universal presence. When no override is supplied, it resolves Event configuration or the Board default and can ultimately use `null`. That makes universal preflight a required runtime follow-up rather than a current guarantee.

11.5 Secrets never become an authoring channel

A BYOK credential can combine `@secret` and `@runtime`:

```
@description("Bring-your-own OpenAI API key for local execution")
@category("Incident AI")
@secret
@runtime
const openAiApiKey: string
```

The declaration is intentionally value-free. FlowScript rejects a non-empty authored secret initializer. Rendering an existing secret omits its stored value, so opening the source, copying it, sending it to FlowPilot, or applying an unrelated edit does not become a secret-read channel.

For a local interactive run, Desktop can prompt once and save the value in the local runtime-value store. The field is masked and can be revealed deliberately. The store is device-local but is not, by itself, a claim of independent encryption at rest; the operating-system account and application data still need protection.

The standard clients filter locally stored secrets out of remote execution payloads. This prevents a browser or Desktop client from casually forwarding a device credential to the hosted executor, but it also means this declaration does not magically make remote BYOK work. An unattended remote Event needs its credential configured through trusted server-side Event configuration. Current Event reads return blank secret values and preserve the stored value when an editor saves an unrelated change; the value is never returned as ordinary configuration. The exact at-rest protection still depends on the deployment and must not be inferred from masking alone.

The OpenAI provider call can then consume the value like any other typed input:

```
const model = ai::provider::openai({
    provider: "OpenAI",
    endpoint: apiBaseUrl,
    apiKey: openAiApiKey,
})
```

This is the right kind of BYOK example because it shows the complete boundary: the Flow author declares a required credential, the runner supplies it through a trusted configuration surface, and the provider node receives it without embedding it in source. It does not imply that every remote execution mode can use the locally stored key.

For hosted OpenAI BYOK, the stronger current path is a private provider/model profile. Its provider credential is stored separately from public model metadata, encrypted by the server, omitted from ordinary profile responses, and hydrated only inside the trusted execution path. This also avoids copying the same API key into every Flow. Offline Desktop execution has a different boundary: it must download the credential into local application settings, so protection of the device profile still matters.

Secret metadata reduces exposure; it cannot make an author-written log safe. There is no universal redaction pass that recognizes every credential copied into an arbitrary message. Never log the key, put it in a ticket, interpolate it into an error, or return it to the caller.

Changing a hidden secret safely

The text editor cannot validate a value it is forbidden to see. If an existing secret has a hidden default, FlowScript will not change its type, container shape, or schema while preserving that unknown value. Declassifying it also cannot replace it with model-authored text in one step. The safe sequence is to clear/declassify through the guarded path, then make the ordinary type or value change explicitly.

These restrictions can feel slower than editing a string. That friction is the point: source and AI tools may modify the credential's contract, but they do not gain the authority to read or write the credential itself.

11.6 Choose the right lifetime

Choose by the cost of losing the value, not by the convenience of its write node. Use this decision table before creating another Flow variable:

Local binding	
Lifetime and ownership	One body path in one run
Use it for	Derived values and readable names
Do not use it for	Cross-run state

Flow variable	
Lifetime and ownership	Shared mutable state inside one run
Use it for	Counters, flags, accumulators
Do not use it for	Durable records

Exposed variable

Lifetime and ownership	Shared App/Board configuration default
Use it for	URLs, limits, feature choices
Do not use it for	Personal secrets

Runtime value

Lifetime and ownership	Local client profile/device configuration
Use it for	Local paths, personal preferences, local BYOK
Do not use it for	Shared unattended remote credentials

Event override

Lifetime and ownership	One configured Event boundary
Use it for	Environment or trigger configuration
Do not use it for	Arbitrary caller mutation

Cache

Lifetime and ownership	Cross-run but disposable, optionally expiring
Use it for	Small hot values and recomputable results
Do not use it for	Audit facts or irreplaceable data

App file

Lifetime and ownership	Durable object/blob storage
Use it for	Documents and large reference datasets
Do not use it for	Highly concurrent field updates

Database/Data Studio

Lifetime and ownership	Durable, queryable records
Use it for	Sync state, entities, history, concurrent work
Do not use it for	Temporary expression results

Chat/session state	
Lifetime and ownership	One interaction context
Use it for	Conversation continuity
Do not use it for	App-wide truth

Cache deserves special discipline. The current key-value cache supports App and user scopes, namespaces, expiration, deletion, and values of roughly one mebibyte or less. It is distinct from the evictable file cache directory. If losing `lastSuccessfulSync` merely causes a safe repeat sync, the key-value cache is reasonable. If losing it could skip work, repeat an irreversible side effect, compromise recovery, or break an audit obligation, store it as a durable database record with the retention policy that obligation requires.

Reference data depends on how it behaves:

- use an App file for a large, versioned, mostly read-only dataset;
- cache a small parsed or frequently accessed derivative that can be recreated; and
- use a database when records are updated, filtered, joined, governed, or changed concurrently.

One application may use all three: a source file, normalized database records, and a cache of the hottest lookup result. “One platform” does not mean “one storage primitive.” It means these mechanisms share the same App, permissions, runtime, and evidence model.

Flow-Like’s current native data path can retain underlying storage versions, but versioning is not a promise of permanent history. Optimization can prune old versions, and teams should set cleanup and retention policy deliberately. If regulation or recovery requires an immutable journal, model that requirement explicitly instead of treating database maintenance history as an audit log.

Chat state is narrower than its friendly names can suggest. A local chat session belongs to one conversation; current “global” chat session state spans chats for the same App/Event on that local client. It is useful for

conversational continuity, not organization-wide, cross-device business truth or security policy.

11.7 Read the complete configuration example

The canonical Chapter 11 fixture keeps configuration and a BYOK key visible without putting any credential value in the repository:

```
@description("Base URL used by the incident AI provider")
@category("Incident AI")
let apiBaseUrl = "https://api.openai.com/v1"

@description("Maximum number of provider attempts")
@category("Incident AI")
let retryLimit = 3

@description("Bring-your-own OpenAI API key for local execution")
@category("Incident AI")
@secret
@runtime
const openAiApiKey: string
```

On the Board, these become typed variables with generated Get/Set operations. The exposed values appear in App configuration. The runtime secret appears in the device-local configuration surface and is absent from rendered source. The provider builder consumes those values through ordinary typed pins.

The fixture deliberately does **not** declare `lastSuccessfulSync` or reference data as global variables. Their absence is part of the design. Good state modeling is often visible in the values you refuse to put into convenient but short-lived storage.

PART II · CHAPTER 12

Functions, Layers, Handlers, and Caching

Turn repeated node graphs into reusable typed functions, distinguish layers from handlers and Events, and design safe cache keys and invalidation.

The first copy of a useful node network feels harmless. The second creates a maintenance question: are these two blocks allowed to evolve independently, or are they supposed to remain the same?

If the answer is “they should stay the same,” they should not be copies. They should be a function.

That is the founder’s practical boundary between visual organization and reusable logic:

> *When you copy a layer because you intend to reuse the same logic, turn it into a function.*

There is a second signal. Flow-Like can automatically reuse a function’s previous result through an explicit cache policy. Ordinary visual grouping cannot carry that callable result contract.

This chapter separates four ideas that can look similar on a canvas:

CONSTRUCT	WHAT IT GIVES THE FLOW
Layer	A nested visual region that keeps one graph readable
Function	A reusable typed boundary inside the same Flow

CONSTRUCT	WHAT IT GIVES THE FLOW
Handler/Event	A triggerable boundary that a caller or agent can invoke
Function cache	An authored promise that a previous output may replace the entire call

Release check: FlowScript functions currently become Function layers with typed boundary pins. Function purity is inferred structurally from their contents; node purity itself is a promise made by each node designer through the presence or absence of Execution pins. Every user function with at least one parameter can be called as a method on its first argument. `@cache` is explicit and may currently be attached even to an impure function; a hit skips its whole body. Cache keys do not include the function body, package versions, global/runtime variables, model choice, or external state unless the author turns those dependencies into inputs. A plain Function layer cannot currently be registered directly as an agent tool: the author must provide a referenceable Event handler. The planned thin automatic Function-to-Event shim is not yet implemented.

12.1 A function is a reusable layer contract

A FlowScript function gives a node network a name, typed inputs, and typed outputs:

```
function normalizeSystemId(systemId: string): (normalized: string) {
    return systemId.trim()
}
```

On the Board, this is not a hidden text routine. It becomes a Function layer. `systemId` is an input boundary pin, `normalized` is an output boundary pin, and every call becomes a Call Function node wired to that layer.

The signature is therefore part of the visible graph contract. Renaming, reordering, or changing the type of a parameter changes a pin. Changing a return changes what callers can consume. That is why a function is more than a tidier rectangle: it establishes one implementation behind all of its callers.

Use a function when at least one of these is true:

- two places need logic that must remain identical;
- the operation deserves a stable typed contract and a meaningful name;
- callers should not depend on the internal node arrangement;

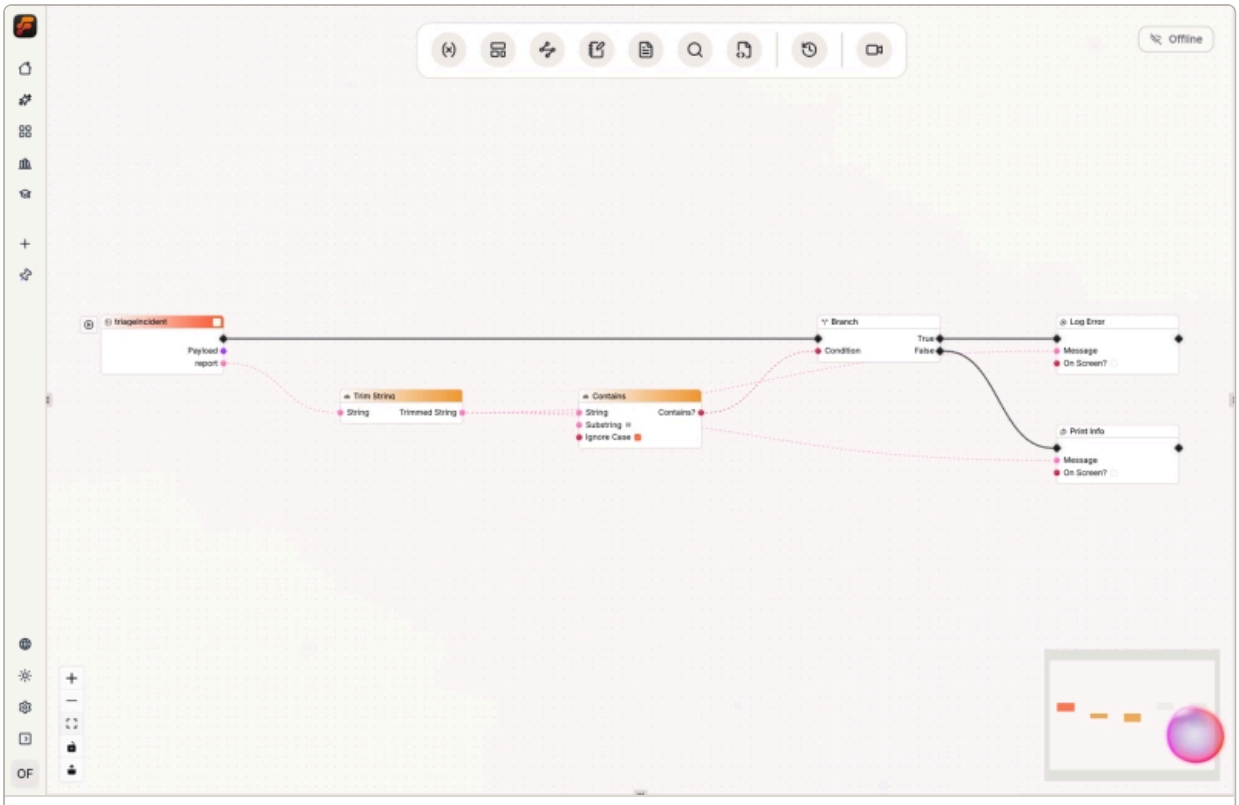
- the result should be testable as a unit; or
- the author wants to apply an explicit cache policy.

Do not extract every cluster reflexively. Five nodes used once may be easier to understand inline, especially when their connections tell the story better than a generic helper name. A function is valuable when its boundary says something more useful than its hidden implementation.

Functions are reusable within their Flow. They are not automatically public APIs, scheduled entries, agent tools, or cross-Flow packages. Those require a triggerable Event boundary or a packaged node, depending on the intended reach.

12.2 Execution moves forward; pure data is pulled backward

Flow-Like has two complementary directions of evaluation:



Execution moves forward on the dark wire; required pure data is pulled through the colored dependency wires before the impure consumer runs.

Execution travels forward through Execution pins. Before an impure node runs, the runtime examines its data inputs. If a required value is produced by pure nodes, it walks backward through those dependencies, evaluates them in dependency order, and then runs the impure consumer.

This gives an apparently small rule a large consequence:

> *An unused pure value performs no work.*

If the output of `systemId.trim()` reaches no return pin and no executing consumer, the runtime has no reason to evaluate it. By contrast, an impure call on the execution chain runs even when nobody uses its data output. Its position on that chain is the reason it exists.

Who decides purity?

The node designer does. At runtime, the structural test is simple:

NODE SHAPE	RUNTIME CLASSIFICATION
No Execution pins	Pure, evaluated when data is demanded
One or more Execution pins	Impure, scheduled through execution flow

That structure is a promise, not a proof. The engine does not inspect a node's Rust or WASM implementation and mathematically establish that it is deterministic or side-effect-free. A node author who hides a network request or mutation behind a pure data pin breaks the execution model.

The design rule is stricter than the shortest academic definition:

- deterministic, inexpensive transformations are good pure nodes;
- operations that may change state are impure; and
- expensive work should usually be impure even when it is deterministic, so its cost and schedule remain visible.

FlowScript derives a Function layer's execution shape from its body. An impure node call, a Board variable write, another impure function, a branch, or a loop gives the Function execution boundary pins. A function made only from pure data dependencies and declared returns can remain pull evaluated.

This means "deterministic" and "structurally pure" are related but not identical. The cached `resolveSystem` example later in this chapter contains an `if`. Its result is deterministic and it has no side effects, but the current planner represents the branch with execution flow, so the Function is structurally impure.

The Unreal Engine comparison

The model is intentionally familiar to Blueprint authors. Epic's official Functions documentation defines a Blueprint Pure function by its promise not to modify state. Impure functions are placed on explicit execution wires; pure functions are evaluated when a connected consumer needs their data. Epic also warns in its UFunctions documentation that pure functions do not cache their results, making non-trivial work a performance concern.

Flow-Like shares the split between explicit execution and demand-driven data. Its recommended node design adds an operational concern: if a calculation is expensive enough that operators should see when and where it runs, making it impure can be the honest choice even when it does not mutate state.

Pure is therefore not a synonym for cached. Pure answers **when is this data needed?** Caching answers **may an older result replace this call?** They are separate decisions.

12.3 Every user function has a receiver

Every user-defined function with at least one parameter can be written as a method on its first argument:

```
const normalized = normalizeSystemId(systemId)
const sameValue = systemId.normalizeSystemId()
```

Both forms call the same Function layer. In the method form, `systemId` fills the first parameter. The remaining arguments follow after it:

```
const { team, runbook } = normalized.resolveSystem(directoryRevision)
```

This is universal for user functions because the first parameter already supplies an unambiguous receiver contract. The editor can offer compatible functions after `receiver.` by checking that parameter's type.

Method syntax does not mean that the function belongs to an object, mutates the receiver, or uses a different runtime mechanism. It is readable call sugar over the same pins. Current Board-to-source projection may normalize an authored method call back to a flat function call, so the spelling is not yet a round-trip identity guarantee.

Catalog nodes have a related but distinct rule. Their node designer selects a receiver pin in the catalog metadata, with a limited first-data-input fallback for compatible namespaces. Not every catalog node automatically becomes a method merely because it has an input.

12.4 Functions are reusable; Events are triggerable

A function has a typed call boundary, but it has no independent runtime entry. An agent cannot trigger a Function layer merely by knowing its name. It needs an Event entry.

The design rule is straightforward: any suitable Event should be explicitly registerable as a tool. When reusable logic already lives in a function, a thin Event shim adapts the function to that trigger boundary:

```
eventsGeneric configureIncidentAgent(payload: Struct, agent: Struct, directoryRevision: string) {
    const configuredAgent = agent::registerFunctionTools({
        agentIn: agent,
        tools: [resolveSystemTool],
    })
    eventsGeneric resolveSystemTool(systemId: string) {
        const normalized = systemId.normalizeSystemId()
        const { team, runbook } = normalized.resolveSystem(directoryRevision)
        return { team: team, runbook: runbook }
    }
    return configuredAgent
}
```

`tools: [resolveSystemTool]` is explicit reference metadata. It records the concrete handler entry the agent may invoke; it is not an ordinary array passed through a data pin. The nested handler is an independent Event entry even though the source places it beside the registration that owns it. Its parameters become the tool's inferred input properties, and its `return` becomes the tool result.

That inference is not yet a complete external contract. The current model-facing schema describes the properties but does not mark them as required, and it does not advertise a declared output schema. The handler remains responsible for validating the values it receives.

The current implementation requires this handler to be written. Registering `resolveSystem` directly is rejected because a Function layer has no trigger node. A future convenience can create the slim shim automatically, but it must still leave the Event and its registration visible.

"Any Event" also needs one release-specific qualification. Current tool registration accepts Event entries marked as referenceable—such as the supported Simple, Generic, Chat, and Widget Action entries—not every arbitrary start node. Registering an Event for one agent does not expose a public API; Chapter 13 covers external App Event registration separately.

Tool schemas are inferred, but policy is still authored. Current function references do not carry their own confirmation, allowed-caller, cost-limit, or permission object. If an operation needs human confirmation or a domain authorization check, put that logic inside or immediately around the handler. Package capabilities and the enclosing runtime permissions still apply, but the Event shim does not invent additional approval on its own.

12.5 `@cache` is an authored promise

Flow-Like never silently decides that a function should be cached. The author opts in visibly:

```
@cache({ namespace: "incident-system-directory-v1", ttlSeconds: 600 })
function resolveSystem(systemId: string, directoryRevision: string): (team: string, runbook: string) {
    let team = "platform-on-call"
    let runbook = "runbooks/general.md"
    if (systemId == "payments") {
        team = "payments-on-call"
        runbook = "runbooks/payments.md"
    }
    return team, runbook
}
```

A cache hit replaces the complete call. The runtime restores the named data outputs, activates the continuation, and runs none of the nodes inside the function. No log, write, API call, ticket, or other side effect in that body happens on a hit.

That leads to the safe contract:

> *Cache a function only when the same declared inputs may legitimately reuse the same outputs for the chosen freshness window—and skipping the entire body is correct.*

The engine currently trusts the author. It permits caching a structurally impure function and the UI can warn about the risk, but Apply does not reject it. This can be valid for an expensive, execution-driven read or deterministic branch. It is not valid for ticket creation, writes, notifications, auditing, or any other behavior that must occur for every call.

What forms the key today

The effective function-cache identity is:

```
App + scope + user when user-scoped + namespace
    + hash(Function layer ID + successfully evaluated input names and values)
```

Object keys are canonicalized and arrays retain their order. The stable Function layer ID prevents two functions in the same namespace from reading each other's entries.

The following are **not** included automatically:

- the function body or output contract;
- catalog, package, or WASM-node versions;
- Flow globals and runtime variables;
- secrets or provider profiles;
- model selection;
- deployment configuration; and
- database, file, or external API state.

If one of those changes the legitimate result, make its identity or revision a declared input, or invalidate/version the namespace when it changes. The `directoryRevision` parameter in the example exists for exactly this reason.

FlowScript's bare `@cache` currently means namespace `global`, a five-minute lifetime, and App scope. The visual Function editor starts from different defaults today, including no expiry. Until those surfaces converge, this book writes the settings-object form with an explicit namespace and TTL. App scope is the canonical FlowScript default and is therefore omitted here. `ttlSeconds: 0` means no expiry and should be reserved for entries with an explicit invalidation and cleanup plan.

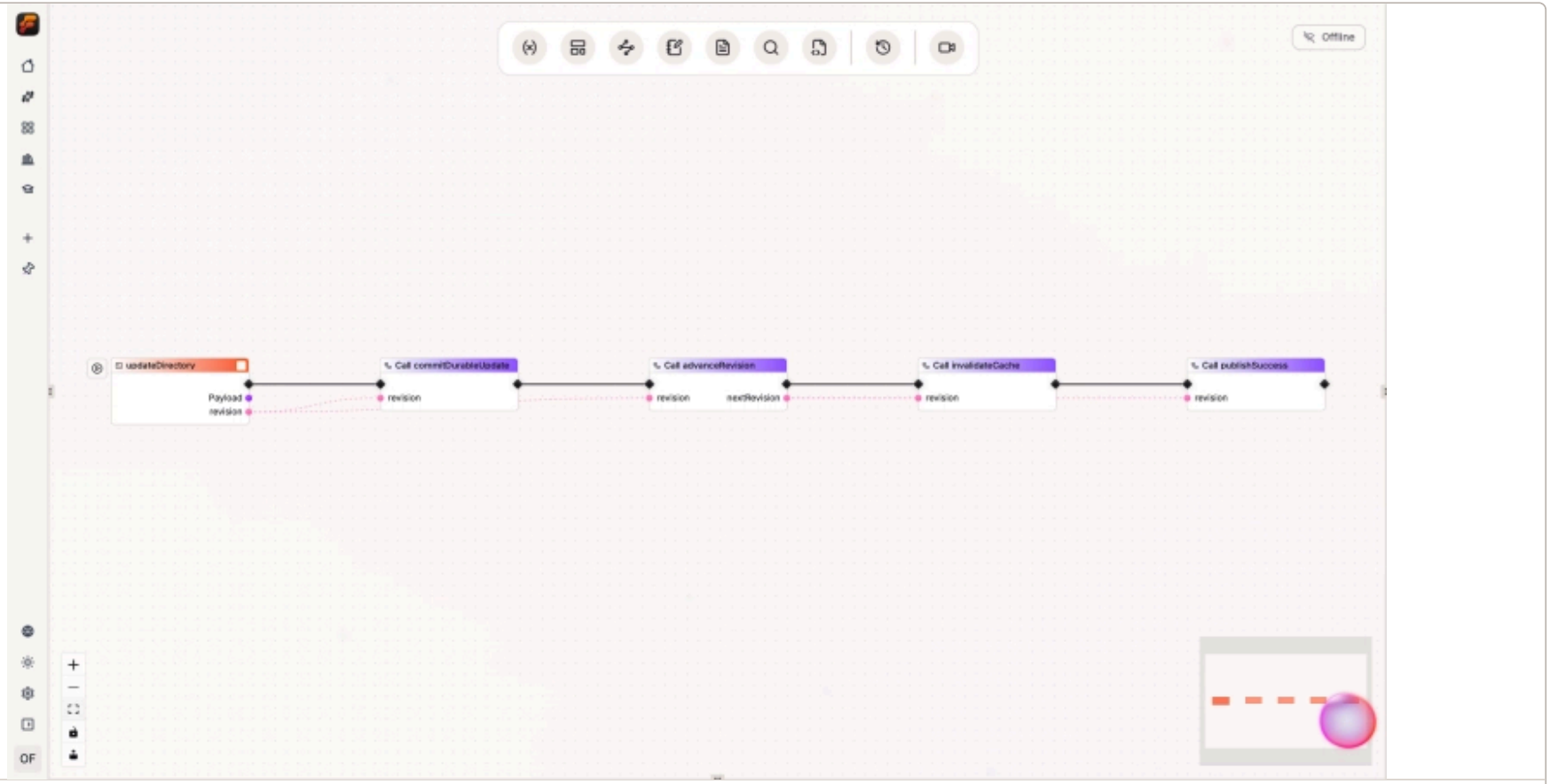
Cache failure is designed to degrade into ordinary execution: an unavailable backend or unusable entry produces a warning and the function runs. A write is best-effort and does not turn a successful Flow into a failure. Concurrent identical misses are not combined into one call, so a cache is neither a distributed lock nor an exactly-once mechanism.

12.6 Invalidate after the source of truth changes

Namespaces give related entries one invalidation boundary. Open the same scope and namespace, then remove the group:

```
eventsGeneric invalidateSystemDirectory(payload: Struct, directoryRevision: string) {
    const directoryCache = data::cache::open({
        scope: "app",
        namespace: "incident-system-directory-v1",
    })
    const deleted = directoryCache.invalidateNamespace()
    log::info({
        message: `Revision ${directoryRevision}: invalidated ${deleted} resolution(s)`,
        toast: false,
    })
    return deleted
}
```

This Event belongs at the successful end of the authoritative system-directory update—not at the end of every read. The safe order is:



This illustrative call topology makes the required order explicit. Production function bodies must implement the durable update, revision, invalidation, and publication contracts they name.

There is still a concurrency edge. A lookup that missed before invalidation can finish afterward and write an old result back into the namespace. Passing `directoryRevision` into the cached function prevents new callers from using that old key. A versioned namespace can provide another clean boundary, although changing a namespace makes old entries unreachable rather than deleting them; TTL or explicit cleanup must eventually remove them.

Changing function code, return names, package versions, model choice, or configuration does not automatically clear old results. That responsibility belongs to the author because only the author knows whether the change alters the meaning of a cached answer.

12.7 Use layers for readability, functions for sameness

An ordinary layer collapses part of one graph into a named nested view. It is valuable when the canvas is crowded, when a section deserves a high-level label, or when a placeholder should define the intended shape before its implementation exists.

A Function layer changes the relationship:

SITUATION	PREFER
One inline section is visually noisy	Ordinary layer
A placeholder sketches work to implement later	Ordinary layer
Copies are intended to remain the same	Function
Several callers need one typed contract	Function
Results should use automatic function caching	Function
A caller or agent must trigger the logic independently	Event, usually wrapping a function

Studio can convert a suitable collapsed layer into a Function while preserving its inner nodes and typed boundary pins, replacing the inline occurrence with a Call Function node. Execution-bearing Functions need one execution entry; pure Functions need none.

The key smell is not visual duplication by itself. Two similar blocks can represent intentionally different domain policies. The smell is copy-and-paste performed with the expectation that future changes must be repeated. That expectation is already a function contract, whether or not the author has named it yet.

12.8 Read the complete function and cache example

The canonical Chapter 12 fixture combines the chapter's boundaries:

```
function normalizeSystemId(systemId: string): (normalized: string) {
    return systemId.trim()
}

@cache({ namespace: "incident-system-directory-v1", ttlSeconds: 600 })
function resolveSystem(systemId: string, directoryRevision: string): (team: string, runbook: string) {
    let team = "platform-on-call"
    let runbook = "runbooks/general.md"
    if (systemId == "payments") {
        team = "payments-on-call"
        runbook = "runbooks/payments.md"
    }
    return team, runbook
}
```

`normalizeSystemId` is pure and runs only when a consumer needs `normalized`. `resolveSystem` is structurally impure in the fixture because it uses a visible `if`, but it is safe to cache: its body has no side effects, its output is determined by declared inputs, and the revision expresses the freshness dependency.

The small in-source mapping stands in for the governed system-directory lookup a production App would perform. The complete fixture also calls the Functions directly, adapts them through an explicit agent Event, and supplies a separate namespace-invalidation Event for the end of the directory's durable update path.

This is the larger pattern: name logic when sameness matters, expose it through an Event when an independent caller needs it, and cache it only when replacing the entire call is an honest thing to do.

PART II · CHAPTER 13

Events, Interfaces, and Complete Apps

Expose one typed Flow through an interactive Page, Quick Action, Cron schedule, and authenticated REST API with explicit identity, versions, and evidence.

A Flow can be correct and still be unusable. Until somebody or something has a supported way to start it, it is an internal implementation.

This chapter turns the incident logic from the preceding chapters into an **Incident Desk App**. The same typed decision becomes available to an App user, a scheduled run, and an authenticated REST caller. We will also give the App a real interface—not a screenshot—and keep every boundary explicit enough to govern, version, and trace.

The result is not one Event with several decorative labels. Different surfaces have different setup contracts. They share the business function while using adapters suited to their callers.

Release check: An event node is a triggerable entry inside a Flow. An App Event binds a compatible entry to a surface, configuration, variable overrides, execution location, and Flow version. Quick Action and Cron can target a Simple Event. A remote REST App Event instead targets a Simple setup entry that registers one or more Generic handler entries. Page actions identify a Board event node and execute it with the current interface state. App roles govern ordinary Event execution and editing; there is no separate per-Event ACL. An Event is Local or Remote—never Hybrid—even when its Board is Hybrid. Numbered Flow versions work today; weighted canary dispatch is represented in the data model but is not wired into the audited execution paths. Remote REST OpenAPI, edge type validation, and rejected-run evidence also have current limitations described below.

13.1 An event node is an entry; an App Event is an exposure

The word *Event* appears at two levels because the platform needs two distinct objects.

An **event node** lives on the Board. It begins execution and defines what the Flow receives. A Simple Event has no data contract of its own. A Generic Event carries a payload and can expose additional typed output pins. Chat and Mail Events provide contracts specialized for those interfaces.

An **App Event** lives around the Flow. It selects:

- the Flow, event node, and either Latest or a numbered Flow version;
- an interface or sink such as Quick Action, REST, Cron, Chat, or Deeplink;
- Local or Remote execution where that surface offers a choice;
- activation, route, exposure, and surface-specific settings;
- variable overrides, including protected secret values; and
- operational release information belonging to that exposed entry.

That separation is useful. One event node may be exposed by several compatible App Events with different schedules, settings, or variable overrides. Conversely, changing a REST route should not require inserting route strings into the business logic.

The memorable formulation is:

> The event node says **what can begin**. The App Event says **how this world may begin it**.

Compatibility still matters. An App Event cannot wrap every event node in every surface. The current product offers Generic Form, API, and Deeplink setups for a Generic Event; Quick Action, API, Cron, Daemon, Deeplink, REST, and MCP setups for a Simple Event; and specialized interfaces for Chat and Mail. The setup screen filters the choices rather than pretending incompatible contracts can be connected.

13.2 Keep one decision and write honest adapters

Our shared decision has a deliberately small result:

```
interface IncidentResult {
    severity: string
    team: string
    runbook: string
}

function decideIncident(
    systemId: string,
    report: string,
    impact: string
): (result: IncidentResult) {
    // Deterministic classification and system-directory lookup
}
```

This Function owns the rule. The interfaces do not copy it. They adapt their input into its three parameters and carry its structured result back to their caller.

The Quick Action adapter is a Simple Event. Its fields come from exposed Board variables:

```
eventsSimple triageQuickAction() {
    const result = decideIncident(systemId, report, impact)
    info({
        message: `Route ${systemId} to ${result.team}; open ${result.runbook}`,
        toast: false
    })
}
```

A Cron App Event can target that same Simple Event. There is no interactive form during a scheduled run, so the scheduled App Event supplies the relevant variable overrides instead. The entry shape is identical even though the caller is not.

REST needs another adapter:

```
eventsGeneric triageRest(
    payload: Struct,
    systemId: string,
    report: string,
    impact: string,
    _client: Struct
) {
    const result = decideIncident(systemId, report, impact)
    return result
}
```

Each named parameter becomes an output pin on the Generic Event entry. The inbound dispatcher fills those pins from the request. `return` publishes the Event result, which the REST surface can turn into a response.

The `_client` parameter is not a user-supplied business field. It carries trusted metadata produced by the selected authentication path, such as verified OAuth claims or a connected-App identity. The Flow may inspect that context and make a finer authorization decision of its own.

Why not register `decideIncident` directly? A Function is a callable layer, not an independently triggerable graph entry. REST registration stores the ID of a concrete event/handler node that the runtime can invoke. `triageRest` is the thin, visible adapter between those two contracts.

SURFACE	BOARD ENTRY	WHERE INPUT COMES FROM
A2UI button	Selected Simple or Generic Event	Current Page element state and invocation context
Quick Action	Simple Event	Exposed variables shown as a form
Cron	Simple Event	App Event configuration and variable overrides
REST route	Generic handler registered by a Simple setup Event	HTTP body, query, headers, and authenticated client metadata

Shared logic belongs in the Function. Surface behavior belongs in the adapter. That small amount of explicit repetition is useful because it keeps a human reviewer from confusing an interactive user action with an unattended schedule or a public network boundary.

13.3 A Page should be experienced, not merely described

The `/triage` experience asks for an affected system, a report, and its business impact. A button then invokes the Page's incident Event and renders its structured result.

The interface below is a real, hydrated React component embedded in this book. Change its values and press **Triage incident**. Its calculation is an explicitly labelled local mock: no Flow-Like runtime or external endpoint is called from the book.

EMBEDDED REACT PROTOTYPE

Incident Desk

Web · Remote

Route an interruption to the people and runbook that can resolve it.

Affected system

PAYMENTS-EU

Business impact

Production stopped

What is happening?

Production is on hold after checkout requests started failing.

Triage incident

This book prototype calculates locally. It does not invoke a Flow.

STRUCTURED RESULT MOCK

Severity SEV-1

Responsible team

payments-on-call

Runbook

`runbooks/payments.md`

Page action `workflow_event → triageQuickAction`

Example output is ready. Change the form to preview another response.

The production Flow-Like Page uses A2UI rather than this book component. Its component tree would contain typed text fields, a select, a button, and result cards. Components whose current values matter to execution are marked as event-relevant. The button's action identifies the event node:

```
{
  "name": "workflow_event",
  "context": {
    "nodeId": "<triage-page-event-node-id>"
  }
}
```

The action context is for routing identity, not an improvised form-data envelope. At invocation, the client combines the current Page elements and relevant input values into the execution payload. The Flow can read the current element values through the A2UI catalog nodes. This keeps the Page model and its behavior connected without hiding a JavaScript callback inside the button.

A Page does not own its navigation path. A UI App Event points to the Page and receives a route such as `/triage`. Today that App interface is an authenticated experience. Anonymous Page hosting is a planned surface, not a guarantee this chapter relies on.

13.4 Quick Action and Cron share a shape, not an identity

The Quick Action can expose the three ordinary variables declared in the fixture:

```
let systemId = "payments"
let report = "Production is on hold"
let impact = "production-stopped"
```

An App user sees those values as fields, changes them, and invokes `triageQuickAction`. A builder can create a second App Event on the same entry for a periodic verification job. That Cron Event can overwrite `systemId`, `report`, or `impact` without editing the Board.

The effective value order is:

1. a permitted invocation/runtime override;
2. the App Event's variable override; and
3. the Board default.

The people who may write the Flow define which variables participate in that configuration contract. A secret variable is handled separately and omitted from ordinary read responses. The REST API key in this chapter has no source value:

```
@description("API key configured on the REST App Event")
@secret
const incidentApiKey: string
```

Its App Event supplies the protected value. The caller's API key proves permission to cross the REST boundary; it is not automatically a credential that nodes should forward to another system. Those are separate trust decisions.

13.5 A REST App Event is a setup program

A single API Event can expose one configured endpoint. The REST surface is more capable and therefore more explicit: the Flow builds a server configuration, registers handlers, attaches authentication, publishes OpenAPI routes, and reaches the REST Server node.

The canonical fixture performs that setup as follows:

```
eventsSimple configureIncidentRest() {
    const base = serverConfig({
        host: "127.0.0.1",
        port: 0,
        tls: { secure: false }
    })
    const routed = base.registerFunction({
        path: "/triage",
        method: "POST",
        fnRefs: [triageRest]
    })
    const secured = routed.registerAuth({
        auth: apiKey({ header: "x-api-key", key: incidentApiKey })
    })
    const documented = secured.registerOpenApi({
        path: "/openapi.json",
        uiPath: "/docs"
    })
    const address = server({ config: documented })
    address {
        onListening: {
            info({ message: "Incident REST surface registered", toast: false })
        }
        onClose: {
            info({ message: "Incident REST surface closed", toast: false })
        }
        execError: {
            warn({ message: "Incident REST setup failed", toast: false })
        }
    }
}
```

`fnRefs: [triageRest]` is reference metadata, not an array passed to an ordinary pin. It tells Register REST Function which triggerable handler to publish. For the current remote path, register one handler per route; the remote setup persists the first reference, while the local socket implementation can invoke several.

Create the App Event with these settings:

1. Target `configureIncidentRest`.
2. Select **REST**, **Remote**, and the intended **Public** or **Internal** exposure.
3. Pin a tested numbered Flow version.
4. Configure the `incidentApiKey` secret override.
5. Choose a stable alias such as `incident-desk`.
6. Save and inspect the setup status and registered routes.

In a remote runtime, REST Server does not bind a long-lived socket inside the setup run. It emits the composed server configuration to the platform. Saving the App Event runs that setup, validates that it reached REST

Server, and persists the resulting route and authentication registrations.

That makes setup part of the release. A newly created REST Event is rolled back when its first setup fails. When an update fails, inbound traffic continues to use the last successful setup version while the error remains visible to the builder. A broken draft does not silently replace a working route.

After a successful setup, the public paths have this shape:

```
POST /r/incident-desk/triage
GET  /r/incident-desk/openapi.json
GET  /r/incident-desk/docs
```

13.6 The HTTP boundary is structured—and still evolving

Call the route with an API key and JSON body:

```
POST /r/incident-desk/triage
x-api-key: <configured secret>
content-type: application/json

{
  "systemId": "payments",
  "report": "Production is on hold",
  "impact": "production-stopped"
}
```

The handler returns one object:

```
{
  "severity": "SEV-1",
  "team": "payments-on-call",
  "runbook": "runbooks/payments.md"
}
```

A plain returned value becomes a `200 application/json` response. Advanced handlers can return a response envelope containing status, headers, content type, and body when the HTTP contract needs more control.

For named handler parameters, query values are collected first and JSON object fields overwrite a same-named query value. The runtime also makes the request, method, path, query, headers, raw body, and `_client` metadata available through reserved handler parameters. Keep the public contract small; accepting the entire request object by default only makes future compatibility harder.

Flow-Like can publish OpenAPI JSON and an interactive browser UI for these registrations. The current remote document reliably describes routes, methods, and authentication, but its request and response bodies remain open objects. The local socket implementation has richer pin-derived request schemas. Neither remote document currently proves that the inbound router enforces the declared Flow types before execution.

That leads to an important separation between design doctrine and the current release:

> **Doctrine:** Reject a request at the earliest boundary where its incompatibility is known, and record the rejection as execution evidence.

Today malformed JSON is rejected before dispatch. Authentication failure and unmatched routes are also rejected at the edge. But field values are not yet validated against the handler's pin schema there; a mismatch may fail later when a node evaluates its typed pin.

Pre-dispatch rejections also do not currently appear as rejected Runs. The persisted Run status model has Pending, Running, Completed, Failed, Cancelled, and Timeout, but no Rejected state. Invalid JSON, failed authentication, and an unmatched route can therefore return before a Run row exists. A type error that reaches execution can appear as a failed Run. The intended design is stronger: create attributable rejection evidence without pretending rejected work executed.

13.7 Identity, permission, and credentials are different boundaries

An ordinary App member needs the App-level **Execute Events** permission to invoke an Event. A builder needs **Write Events** to create, activate, reconfigure, or repoint one. Owners and admins inherit these abilities, and a custom role may receive them. There is currently no separate ACL on each App Event.

The Flow can still make a domain decision after entry. User-context nodes expose the executing subject, role, permissions, and role attributes, so the graph can ask questions such as "may this caller triage payment incidents?" That check is application logic and remains visually reviewable. The core context also models arbitrary custom attributes, although the normal audited API path does not yet populate all of them.

REST adds another boundary:

CONCERN	WHAT ANSWERS IT
May this App member invoke ordinary Events?	App role and `ExecuteEvents`
May this network caller enter the REST route?	Configured REST authentication
Which connected App called an Internal route?	Connected-App proxy identity and role
May this caller perform the domain action?	Explicit checks inside the Flow
Which credentials may nodes use downstream?	Caller or App Event/sink credentials, chosen for that execution setup

Public REST supports no authentication, API key, static bearer token, Basic authentication, HMAC-SHA256, and OAuth/OIDC bearer validation. This exercise deliberately uses an API key. Leaving Register REST Auth disconnected creates an unauthenticated surface and should be an explicit, reviewed decision—not an accidental default.

API-key possession does not identify a person. Verified OAuth claims can contribute subject, issuer, audience, scopes, and related metadata to `_client`. An Internal REST Event is reachable through an approved App connection rather than the public router, but that connection does not bypass authentication configured on the REST registration.

Interactive invocations can execute with the signed-in caller's credentials where configured. Unattended and public Event sinks normally use credentials deliberately stored with the App Event or sink. Platform storage

credentials are scoped separately. Treating all three as “the user’s token” would erase the very boundary the App needs to audit.

13.8 Hybrid is a Board choice, not an Event location

Boards support Local, Remote, and Hybrid execution modes. App Events support only Local or Remote.

- A Local Board forces its App Events local.
- A Remote Board forces them remote.
- A Hybrid Board preserves the Local or Remote choice made for each App Event.
- The web client dispatches through the remote backend; Desktop and Studio can execute locally.
- A remote REST App Event is Remote by definition.

The product-level reason for Hybrid is simple: builders should be able to work locally in Studio while the web experience runs on governed remote infrastructure. That does not mean half of one invocation runs on a laptop and the other half in the cloud. One execution has one location. Nodes inside it may call remote systems, but the Flow run itself is not teleported between runtimes.

13.9 Pin the contract before other people depend on it

An App Event can follow **Latest** or target a numbered immutable Flow version. Latest is useful while authoring because every saved Board change is immediately exercised. It is a poor default for a production API whose callers expect stability.

Use this release sequence:

1. Develop and test against Latest.
2. Save a numbered Flow version after the Page, Quick Action, and REST handler pass their cases.
3. Point the production App Events at that version.

4. Run REST setup and verify its route, authentication, OpenAPI document, and one failure case.
5. Promote the change through the organization's admin/builder review.
6. Keep the prior version available for an explicit rollback.

A caller cannot override the App Event's configured Flow version. REST registrations additionally have an App Event setup version and a pointer to the last successful setup. These identifiers solve different problems: the Flow version freezes authored logic; the setup version selects the published route configuration.

Do not describe weighted canary rollout as a current guarantee. The App Event model can store a canary target and weight, but the audited invocation paths select the primary target and do not perform weighted routing. Canary configuration is product intent until that selection is executed and tested end to end.

13.10 The complete App checklist

The Incident Desk is complete enough to hand to another team when all of these questions have answers:

- **Entry:** Which Board event begins each use case?
- **Contract:** What typed values enter, and what structured value returns?
- **Interface:** Is the caller using a Page, Quick Action, schedule, REST route, or another surface?
- **Identity:** Who or what is calling, and what further domain authorization is required?
- **Credentials:** Which secrets may this execution use after entry?
- **Location:** Is this App Event Local or Remote?
- **Release:** Which immutable Flow version and successful setup are live?
- **Evidence:** Where does a successful run, failed run, or pre-run rejection appear?

The last item intentionally exposes a current gap instead of concealing it. A complete platform should preserve the evidence for a rejected request as deliberately as it preserves a failed node.

The larger pattern is already in place. The React prototype makes the intended experience tangible. A2UI keeps the production interface structured. The Quick Action and Cron adapters reuse the same decision without pretending their callers are alike. The REST setup makes routes, authentication, documentation, and release behavior visible. And every execution that reaches the Flow still returns to the same graph where an operator can identify the responsible block.

That is the step from a callable Flow to an App: not more hidden framework code, but explicit boundaries around logic that remains readable from both sides.

PART III

The Two-Way Contract

Open the machinery that keeps the visual Board and canonical FlowScript aligned through typed meaning and guarded changes.

PART III · CHAPTER 14

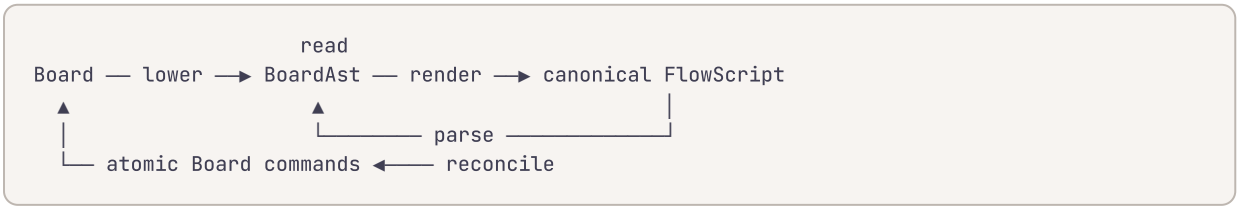
Board ⇄ AST ⇄ Text

Learn how Flow-Like lowers a Board to a typed AST, renders canonical FlowScript, reconciles semantic edits, applies atomic patches, and preserves identity.

Two editable views create a harder problem than two read-only views.

If FlowScript were only an export, Flow-Like could print a Board and stop. If the Board were only a diagram generated from code, Flow-Like could throw the old graph away after every text edit and draw a new one. Neither shortcut would satisfy the promise of equal authoring surfaces. A builder must be able to move one node without replacing the program, while a developer must be able to change one literal without erasing the builder's layout.

Flow-Like solves that problem with a typed intermediate representation called `BoardAst`. The AST captures what the program *means* without pretending that source text should own every visual and runtime detail of the Board.



The persisted Board remains the executable model. The AST is the temporary semantic contract. FlowScript and the visual editor are equal ways to author that model.

Release check: The current FlowScript editor treats text changes as a draft until Apply. Studio/Desktop can also run the authoritative reconciler to show a semantic command preview; web currently has client-side structural feedback and the authoritative server Apply response, but no equivalent pre-Apply command preview. Parsing validates syntax; reconciliation resolves catalog calls and validates the proposed graph change. Any diagnostic blocks Apply. A valid edit becomes a minimal batch of Board commands, applied with rollback on failure, and the editor reloads canonical source from the resulting Board. Formatting choices are deliberately standardized rather than preserved. Hidden secret-variable values do not enter rendered source and survive unrelated edits. Existing anchored Function signatures cannot yet be changed safely through FlowScript; the edit is rejected instead of partially rebuilding the boundary.

14.1 One program, three representations

The three forms overlap, but they do not own identical information.

REPRESENTATION	ITS JOB	INFORMATION IT OWNS
Board	Persist and execute the Flow	Node and pin identity, connections, defaults, layers, coordinates, viewport, presentation, package and runtime metadata
`BoardAst`	Express the typed meaning shared by both editors	Imports, interfaces, variables, Functions, Events, modules, detached chains, ordered statements and expressions, plus identity anchors used during editing
FlowScript	Give people and tools a compact authoring surface	A canonical textual projection of the AST's program structure

This is why "both views are the source of truth" is close but imprecise. It sounds as if Flow-Like stores two independent programs and later tries to synchronize them. It does not. The sharper contract is:

> *Studio and FlowScript are equal authoring surfaces over one underlying Flow model.*

An edit on the Board changes that model and therefore changes its next textual projection. An applied text edit becomes Board commands and therefore changes the next visual projection. The views agree on executable meaning, not on every character or pixel.

That distinction matters in practice. A Board node needs an opaque identity so wires, logs, undo, and versions can keep referring to the same operation. A programmer needs a readable name such as `productionStopped`. Both describe one operation, but the readable name is not a sufficient database key and the database key is not a good programming language.

14.2 Why Board JSON is not the language

Try this small experiment in Studio:

1. Select two or three connected nodes on a Board.
2. Copy them.
3. Paste the clipboard into a text editor instead of another Board.

What appears is useful JSON. It contains serialized nodes, comments, cursor position, layers, variables, schema references, pin IDs, defaults, and connection sets. Flow-Like can paste that data back because it is an excellent transport format. It can mint new node, pin, and layer IDs, then reconnect the copied selection.

Now imagine reviewing a five-hundred-node application in that form. The business decision is buried among storage identities, adjacency lists, coordinates, compatibility data, and UI metadata. Moving a node changes the document even when the program means exactly the same thing. An AI editing that JSON must manipulate graph bookkeeping correctly before it can express a simple domain change.

This is where the AST became necessary.

FlowScript was already in mind while Boards were being built, but the architectural need crystallized later: once text became an authoritative editing surface, particularly a safe surface for FlowPilot, “print this graph readably” was no longer enough. Direct text-to-Board conversion would couple language syntax to persistence and encourage whole-graph reconstruction. A typed semantic pivot instead lets both directions meet on the program’s meaning.

The repository history supports that interpretation. The first FlowScript implementation landed with the FlowPilot update and introduced the AST, parser, renderer, Board lowering, and reconciliation together. That is an architectural clue, not a claim that every design decision was made in one moment.

The practical outcome is more important than the chronology:

- Board JSON remains optimized for persistence, execution, transport, and the visual editor.
- FlowScript remains optimized for reading, reasoning, completion, review, and focused edits.
- `BoardAst` prevents either representation from dictating the other's accidental details.

14.3 Lowering a Board extracts the program

Lowering begins with the live Board and its catalog. It does not simply iterate through a node map and print one line per entry. A graph has connections; a program has scopes, expressions, statements, and order. The lowerer must recover those language structures.

It performs several related jobs:

- Board variables become typed top-level declarations.
- schema-backed shapes become FlowScript interfaces;
- Function layers become typed `function` declarations;
- trigger nodes and their reachable execution paths become Event bodies;
- every execution chain no trigger reaches becomes its own `detached` block;
- impure nodes on execution wires become ordered statements;
- pure data dependencies are pulled backward and nested as expressions;
- named execution outputs become branches such as `onSuccess`, `execError`, or an `if` body; and
- node, variable, and layer identities can become editing anchors.

That `detached` block exists because the chain has to go somewhere. FlowScript has no top-level statement position, and the chain's root is an ordinary node, so spelling it as an entry would misrepresent it and hide its inputs. Nothing inside such a block runs; it reports what the Board already contains rather than offering a way to author new work.

Consider the Incident Desk rule:

```
const productionStopped = report.contains({
    substring: "production is on hold",
    ignoreCase: true
})
if (productionStopped || customerFacing) {
    severity = "SEV-1"
}
```

The Board does not store that exact syntax. It stores a String Contains operation, variable reads, a boolean operation, a Branch, a Set Variable operation, pins, and the relevant data and execution connections. Lowering recognizes that shape and chooses the compact expression and control-flow forms.

Some visual helpers disappear from the semantic projection. A reroute helps a wire travel cleanly across a canvas but performs no domain operation, so lowering follows through it. Collapsed visual boundaries may flatten where their contents remain the real computation. Coordinates can provide a stable ordering hint for otherwise independent entries without becoming source-code syntax.

This is semantic compression, not data loss from the Board. The live Board still exists while the text is being edited.

14.4 Rendering chooses one canonical spelling

Once lowering has produced `BoardAst`, rendering is deterministic. The renderer chooses a stable document order, four-space indentation, double-quoted strings, decorator order, spacing, parentheses, and declaration layout.

Canonical source gives humans and machines a smaller comparison surface. Two authors cannot spend an afternoon debating semicolons or quote style because those choices are not part of the contract. After Apply, the system reloads source from the Board and standardizes it as much as possible.

For example, an author may type:

```
if (customerFacing && impact == 'production-stopped') {
    severity = 'SEV-1';
}
```

Pure parse-and-format produces the canonical form:

```
if (customerFacing && (impact == "production-stopped")) {
    severity = "SEV-1"
}
```

The added parentheses make precedence explicit. Quote choice and semicolons disappear. This is a semantic AST, not a concrete syntax tree intended to reproduce the author's keystrokes.

Board lowering also chooses graph-derived sugar. Pure node outputs may become nested expressions. Catalog metadata can turn a static call into a method call. Imports are derived only when they make several calls clearer without creating ambiguity.

Suppose two namespaces both expose `contains`. Opening both names would make the call unclear, so the lowerer keeps the conflicting use inline:

```
const inReport = string::contains({
    string: report,
    substring: "production is on hold",
    ignoreCase: true
})
const inSystems = array::contains({ array: affectedSystems, value: systemId })
```

Flow-Like does not guess which `contains` the author meant. It uses the qualified spelling until an import is safe.

Secrets do not need a readable value

A secret variable can have a configured value on the Board while its textual declaration has no initializer:

```
@secret
const incidentApiKey: string
```

The secret value is deliberately omitted when the Board becomes an AST. Because an unchanged declaration carries no replacement value, applying an unrelated edit preserves the configured secret rather than clearing it. FlowScript also refuses to populate a new secret from a nonempty source initializer.

This protection is specific, not magical. It does not make arbitrary downstream values safe to log or return, and it should not be generalized into a claim that every sensitive pin on every local and remote rendering path has been audited. Secret handling still depends on the node, execution boundary, and storage path involved.

14.5 Parsing turns a draft back into typed meaning

Typing in the editor does not mutate the Board on every keystroke. The text buffer is a draft. Flow-Like parses it for feedback, and changes the Board only when a valid revision is applied. Studio/Desktop can additionally reconcile the draft against a cloned Board for an authoritative command preview without persistence or undo mutation. Web currently performs that authoritative step in the server Apply path.

Parsing answers questions such as:

- Are delimiters balanced?
- Is this declaration allowed in this scope?
- Does the expression have a valid syntactic shape?
- Is a decorator written correctly?
- Is the document nested within the defensive parser limit?

It deliberately does not answer every program question. The parser can recognize the shape of `madeUp::operation({ value: 1 })` without knowing whether the installed catalog contains that node. It can recognize a named argument without knowing whether the node owns a compatible pin. Those checks require the live catalog and Board, so they belong to reconciliation.

The editor therefore has two useful feedback layers:

1. **Parse feedback** identifies the first syntactic problem with a line and column.
2. **Reconcile feedback**, when the client requests that authoritative check or Apply reaches the server, reports unresolved calls, ambiguous names, missing pins, incompatible types or shapes, invalid boundaries, policy limits, and unsafe edits.

Only a revision that passes both layers can be applied. Invalid source remains a harmless draft, and the author gets a concrete problem to fix.

For experts: the small parser behind the editor

The current parser is hand-written and intentionally compact:

```
characters
  → context-free tokens
  → recursive-descent declarations and statements
  → Pratt-parsed expressions
  → typed BoardAst
```

The lexer records line, column, and byte position. Recursive descent handles document structure; a Pratt parser handles operator precedence and associativity. A shared nesting budget of 128 protects blocks, modules, expressions, and nested interface types from adversarial editor or API input.

The diagnostic model is still modest. Parsing stops at its first error and reports a line and column rather than a rich multi-error span set. Template-expression errors receive rebased source coordinates, while later semantic diagnostics locate relevant tokens on a best-effort basis. This is enough for a useful guarded editor, but it should not be described as a finished compiler diagnostics system.

The formatter's invariant is straightforward:

```
canonical = render(parse(authored))
render(parse(canonical)) == canonical
```

That stable second rendering is more important to Flow-Like than preserving arbitrary first-pass style.

14.6 Reconciliation plans the smallest valid change

After parsing, the reconciler compares the edited AST with the current Board. It uses the catalog to resolve calls, derives their expected pins, checks types and shapes, accounts for dynamic pins, and matches existing entities through stable identity information.

Its result is not a replacement Board. It is a command plan using the same vocabulary as the visual editor, including operations such as:

- update one node pin;
- add, update, or remove a node;
- connect or disconnect two pins;
- add or update a variable;
- add, update, or remove a layer; and
- move a node into another layer.

Canvas movement also uses the shared command vocabulary, but FlowScript reconciliation currently does not reposition existing nodes. New text-created structures receive automatic placement while untouched coordinates remain untouched.

That command boundary is the central preservation mechanism. If one literal changes, the plan can update one pin while leaving every other node ID, coordinate, comment, connection, package field, and visual choice alone. If a new expression requires three nodes and four wires, the preview can say so before any command runs.

The Studio/Desktop Apply preview groups the semantic consequences so a reviewer can distinguish an ordinary configuration update from a structural or destructive edit. Web currently returns the same class of authoritative diagnostics and command outcome during Apply rather than showing that command plan beforehand. Deletions need separate approval, which Chapter 15 examines in detail.

Apply then follows a fail-closed sequence:

1. Reconcile the complete draft against the current Board and catalog.
2. Stop with no executed commands if any diagnostic exists.
3. Stop if a destructive plan has not received explicit approval.
4. Validate and execute the command batch against a staged Board.
5. Roll back earlier commands if a later command fails.
6. Persist the Board only after the complete application succeeds.
7. Lower and render the result back into canonical FlowScript.

The last step matters. Successful Apply does not preserve the author's preferred spelling; it preserves the accepted program and the live Board information outside that program.

14.7 Follow one Incident Desk change through the loop

Start with the anchored line that recognizes the report from the 3 A.M. call. The identity below is shortened for the book:

```
const productionStopped = report.contains({
    substring: "production is on hold",
    ignoreCase: true
}) //@n:contains-report
```

Change only the literal:

```
const productionStopped = report.contains({
    substring: "customer-facing outage",
    ignoreCase: true
}) //@n:contains-report
```

The parser produces the same statement shape with a different string. The anchor identifies the existing Contains node. Reconciliation sees that its `substring` input is not connected to another node and that only its stored value changed. The resulting plan contains exactly one `UpdateNodePin`; it does not remove and recreate the node.

That is the smallest useful demonstration of the contract:

```
one source literal
    → one AST literal
    → one changed input pin
    → one Board command
    → the same node in the same place
```

A structural edit makes the same principle more visible. Derive one more signal from the existing report without changing the Function boundary:

```
const customerFacing = report.contains({
    substring: "customer-facing",
    ignoreCase: true
})
```

Then revise the rule:

```
- if (productionStopped) {
+ if (productionStopped || customerFacing) {
      severity = "SEV-1"
  }
```

For the current standard catalog, the semantic delta is a new String Contains operation, a Boolean OR operation, their literal settings, and the data wires into the existing Branch. Unchanged nodes retain their identity and placement. The exact preview count still depends on the installed catalog and live graph shape, which is why the product shows the real plan rather than promising a fixed expansion in prose.

The example intentionally avoids adding `customerFacing` as a parameter to the existing anchored `decideIncident` Function. The current reconciler rejects signature changes to an established Function boundary because it cannot yet migrate that layer and all callers safely. A new Function can

be created with the desired signature; changing an existing one requires a future safe migration path. Failing closed is better than silently detaching caller pins.

14.8 The text does not own the canvas

Suppose a domain expert places the severity branch beside the input it explains, colors its layer, adds a Board comment, and routes a long connection around an unrelated cluster. None of those choices needs a FlowScript keyword.

Because Apply patches the live Board instead of reconstructing it, an unrelated text edit can preserve:

- node and pin IDs;
- coordinates and viewport;
- layer colors, icons, and presentation settings;
- package, score, version, and runtime metadata;
- decorative Board comments; and
- reroutes that make wires readable.

Preservation does not mean every visual construct has an equivalent line of source. A reroute may vanish while reading FlowScript because it is semantically transparent, then remain on the live Board because no command touched it. Board comments are not currently lowered into FlowScript comments, and source comments are not a reliable way to create or replace decorative Board comments. A complete text-only graph rebuild therefore cannot promise pixel-identical recovery.

This leads to the right definition of round-trip fidelity:

> *FlowScript aims to preserve supported program meaning and untouched Board identity, not arbitrary author formatting and not a byte-for-byte or pixel-for-pixel serialization.*

There are still unsupported edges. Existing Function signature migration is one. A whole-Board unchanged round trip also has known difficult fixtures around duplicated handler names and cross-handler references. Report such cases with the Flow-Like version, the smallest reproducing Board, and the text before and after Apply. “Equal authoring surfaces” is a contract strengthened by tests, not a reason to hide unfinished cases.

14.9 The mental model to keep

When you edit the Board, Flow-Like owns graph identity and projects readable source. When you edit FlowScript, you are proposing a semantic patch to that graph. The AST is the meeting point that makes both operations precise.

Remember the six verbs:

1. **Lower** the Board into typed meaning.
2. **Render** that meaning as canonical FlowScript.
3. **Parse** an edited draft back into typed meaning.
4. **Reconcile** its semantic difference against the live Board and catalog.
5. **Apply** the minimal valid command batch atomically.
6. **Reload** the result so both editors show the same accepted program.

The next chapter looks at the identity anchors, correction proposals, deletion guards, and scoped editing rules that make step four safe on large Flows.

FlowBook · The FlowScript Book

Open edition · 2026 · © 2026 Flow-Like

book.flow-like.com