FLOW-LIKE

ONE EXECUTABLE FLOW

FlowBook

A Developer's Guide to Flow-Like

Build reliable software as typed text and 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 how Flow-Like connects typed FlowScript, a visual workflow, existing systems, and governed execution in one application model.

FlowBook is the book you are reading. **FlowScript** is Flow-Like's typed text form for workflow logic. A **Flow** is that executable logic, whether you inspect it as text or as a visual graph.

The distinction matters because most companies are extending an estate that already exists. Customer data lives in current databases. Core processes cross older services, SaaS products, files, queues, and devices. New software has to join that estate without making it harder to operate.

AI raises the rate of change. A useful prototype can now arrive in days, sometimes hours. The hard question comes later: can the company operate it safely after the original author has moved on? Faster generation increases the value of a shared application model.

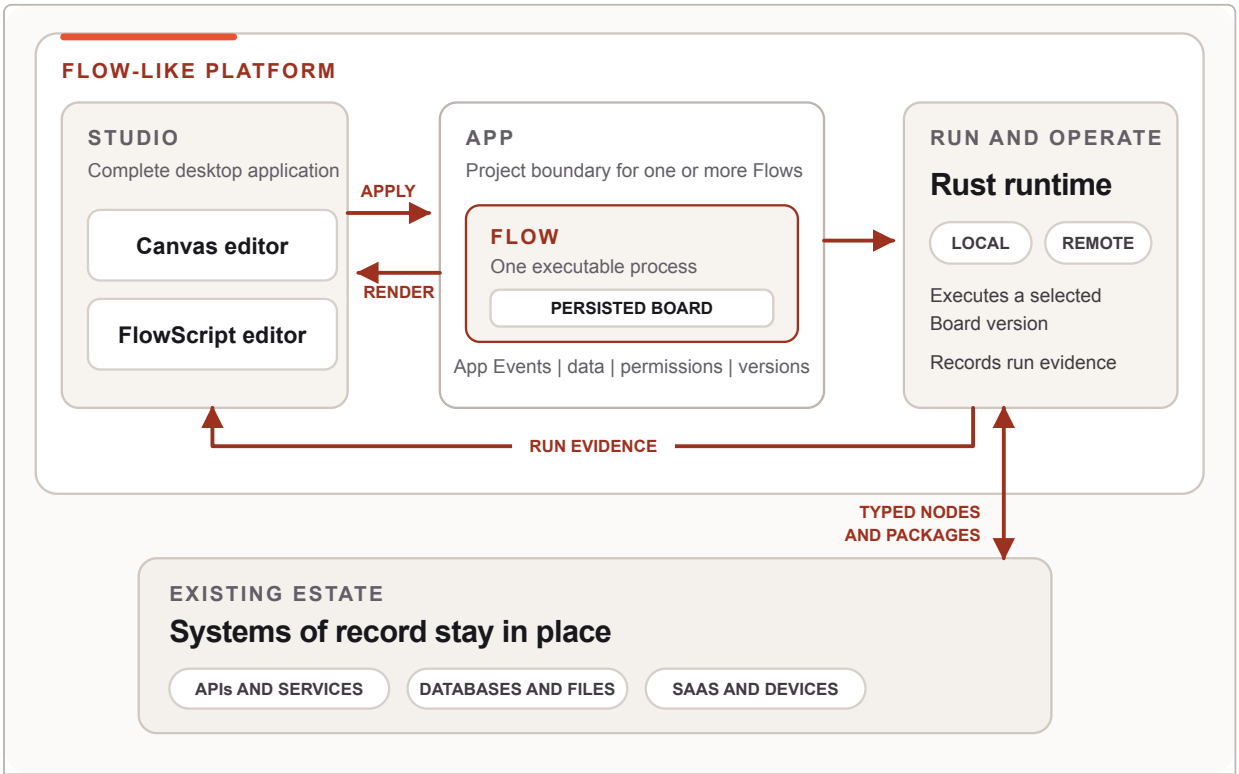
Flow-Like begins with a practical expectation:

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

The program should retain visible operations, typed boundaries, execution order, permissions, versions, and failure evidence after it leaves its author's editor. That expectation guides the language, the visual workflow, and the platform around them.

The map

The following picture is enough to place the terms used throughout the book.



Studio is the complete desktop application. It edits Apps and Flows; a local or remote runtime executes the selected Board version. Typed nodes and packages reach systems of record that remain in place.

TERM	MEANING IN THIS BOOK
Flow-Like	The platform that supplies authoring, execution, data, interfaces, packages, permissions, and operational evidence.
Studio	The complete desktop application. It includes Flow editing, Data Studio, packages, run inspection, and other App tooling.
App	The project and governance boundary for related Flows, data, Events, members, and release settings.
Flow	One executable workflow inside an App.
Canvas / Board	The visual graph of a Flow. *Canvas* names the editor; *Board* is the precise name for the persisted graph model.
FlowScript	The typed text view used to author the same Flow logic.
Runtime	The Rust execution system that runs the Board and records evidence about the run.

When this book says “open it in Studio,” it means the desktop application. When it refers to a node on the canvas or a change to the Board, it means the visual representation of the logic. FlowScript always refers to the textual representation.

Why the company should care

An AI-first company will encode more decisions and routine work in software. If every team builds that logic with its own prompt scripts, credentials, deployment conventions, and logs, the operating model fragments while the amount of software grows. Prototype speed then creates more systems that only their authors can safely change.

Flow-Like puts Apps, deterministic logic, model calls, data access, permissions, versions, and run evidence inside one inspectable model. Teams still choose architectures and operating policy. The platform gives those choices a common place to live and a consistent path to execution.

Adoption can begin with one process. A company might place a fragile integration, a new AI classification step, or an internal tool in a Flow while the current database and services stay in place. The Flow calls those systems through typed nodes or packages, and existing callers can reach it through an Event or API. Over time, more logic can move into visible Flows where that improves ownership and operation. A rewrite is not the entry fee.

For a technical decision-maker, that is the reason to keep reading: the platform offers a way to increase software output without multiplying private execution models. The rest of this book tests that claim at developer depth.

Two views of one program

On the canvas, a developer can see where data comes from, how execution branches, and which node produced a result or failure. That map is valuable during design, review, and incident response.

Text supports a different kind of work. It is dense, searchable, easy to compare, and efficient when a Flow contains many operations. It also works well with language tooling and coding agents.

Inside Studio, the canvas and the FlowScript editor act on one Flow model. Change the graph and the source changes. Apply a FlowScript edit and Studio reconciles it into Board operations. Neither view is a separately maintained diagram.

> **THE CURRENT TECHNICAL CONTRACT**
>
> FlowScript is an authoring language. Its text is parsed and reconciled into the persisted Board behind a Flow, and the Rust runtime executes that graph. The text and canvas have authoring authority over one program; they are not independent execution engines.

This contract avoids a familiar source of drift. A diagram can be correct on the day it is drawn and wrong after the next release. Here, the visual structure is part of the program that runs. Runtime evidence can point back to the node responsible for it.

Why FlowScript exists

A useful workflow platform needs more than a few high-level boxes. Real applications transform values, branch, iterate, manage state, call services, and compose reusable operations. Exposing those details keeps the graph honest, although a large graph takes time to navigate and edit.

Some workflow tools deal with scale by placing a JavaScript or Python editor inside one node. The node stays small while a second application grows behind its label. Dependencies, side effects, and error paths become harder to inspect from the graph.

FlowScript provides the density of text while preserving Flow-Like's typed building blocks. Its calls, expressions, branches, loops, functions, and Events reconcile to nodes and connections the canvas can show. A developer can move to text for a precise change and return to the graph for locality or run evidence.

When the catalog lacks a reusable capability, a package can add scoped WebAssembly nodes with declared contracts. The new capability then becomes a visible building block for later authors. This extension model gives developers room to solve specialized problems without placing an uninspectable program inside a generic code box.

Where AI fits

Once the application model is clear, AI has two natural places in it.

FlowPilot or another coding agent can author a change. It may propose FlowScript, assemble nodes, or adapt an existing App. The proposal passes through the same declarations, type checks, reconciliation, and review path as a human edit. An agent receives useful constraints instead of privileged access to a hidden shortcut.

A model can also perform one operation during a run. Classification, extraction, summarization, and response generation are reasonable examples because they involve semantic uncertainty. The model call should appear as an explicit node with narrow inputs, a typed result, an error path, and usage evidence. Deterministic nodes prepare and validate the data around it.

Exact work stays deterministic. A probabilistic call adds no value when the outcome is already defined, as with arithmetic, schema validation, identifier handling, or a known routing rule. Keeping the uncertain boundary small lowers cost and latency and gives failures a specific place in the Flow.

Later chapters return to model selection, usage controls, and AI-authored changes after the core Flow model is familiar. That sequence keeps AI inside the architecture instead of turning it into a separate topic with its own vocabulary and runtime assumptions.

Who this book serves

FlowBook is written primarily for developers. It assumes basic programming familiarity and explains the Flow model before opening parser, runtime, and deployment internals. TypeScript experience will make parts of

the syntax familiar, but it is not required.

Domain experts, operators, and technical leaders can follow the same path through the worked applications. The canvas gives them a way to inspect the process without requiring a second, manually maintained explanation. Chapters that focus on language internals can be skipped and rejoined later.

What we will build

The first application is **Incident Triage**, a deterministic Flow that receives a report, normalizes it, classifies severity, branches, and records the result. It needs no model, account, API key, or external service. You will edit it from both views and break it on purpose so the run evidence has something useful to show.

Each language feature then stays tied to a concrete problem, its FlowScript form, the graph it represents, and the evidence left by execution. The later **Incident Room** application combines those ideas in a private document assistant and introduces model calls only where the task needs semantic judgment.

The book names release boundaries when they matter. A construct that does not round-trip safely will be identified. Deployment policy will not be presented as a language guarantee. Performance claims will stay attached to the workload that produced them.

Chapter 1 begins with the incident that made this level of visibility feel necessary: a room full of capable people waiting for the only person who understood the failing system.

PART I

Software That Explains Itself

Begin with the founding incident, establish the operating principles, and build the first Flow through both authoring views.

PART I · CHAPTER 01

The 3 A.M. Call

A major incident reveals why software needs visible structure, operational evidence, and domain knowledge that survives its author.

At three in the morning, every minute feels expensive. On this call, it was. An important chain of systems inside a large enterprise had failed, and each hour of downtime risked millions in damage.

The domain experts could give us one concrete fact: production was on hold. Nothing in front of us showed the technical path that led there.

I did not know the affected systems, their interfaces, or the decisions behind them. That is normal in a large organization. An incident call brings the necessary knowledge into one place. This time, the knowledge we needed had not arrived, and the system could not supply it.

1.1 Waiting for context

We were trying to reach the person who knew the affected system well enough to tell us where to look. Until then, the rest of the room inspected fragments and exchanged partial theories. The failure was serious, yet investigation depended on locating one human being in the middle of the night.

Hours passed.

Complex systems fail. Dependencies disappear, interfaces return surprising data, and earlier assumptions meet conditions their authors never saw. Reliability has to account for those events. What troubled me was the blindness that followed this one.

The organization had engineers, operators, escalation paths, and an active incident process. A responder still could not follow the execution far enough to identify the responsible operation. The scarce resource was context.

The system expert carried that context: which component mattered, what its interface expected, and which behavior pointed to the fault. The outage contained a technical failure and an explanatory failure. The second one kept a capable room idle while the first continued to cost money.

1.2 The explanation had drifted away

Incident reviews often answer this problem with more documentation. A better diagram, another runbook section, or a linked architecture decision can all help. Each also describes the system from a particular moment and viewpoint.

The running application keeps changing. A diagram can remain plausible after the execution path has moved. A runbook covers failures its authors anticipated. A ticket records work for the people who performed it. Under delivery pressure, teams update the artifact that makes the system run; secondary explanations depend on time and memory.

Source code stays close to the implementation, although it may still leave an unfamiliar operator to reconstruct configuration, infrastructure, and ownership across several repositories. Logs can name an error without locating it in the larger process. The architecture exists, scattered among code, deployment settings, dashboards, tickets, and people.

I wanted the executing system to carry more of that explanation. Its real operations and boundaries should remain visible. Runtime evidence should point to the operation that produced it. A responder should have a useful starting point before the original author joins the call.

1.3 Start at the failed operation

Imagine the same incident represented as a visible Flow. The canvas shows the operations and their connections. Inputs and outputs have types. External calls appear as identifiable nodes. Run evidence belongs

to the building block that emitted it.

One node shows the failure.

The graph cannot repair an unavailable dependency or choose the correct business response. It can change the first hour of investigation. The team opens the failed node, inspects what reached it, and follows the surrounding path. Types narrow the plausible causes. Per-node logs connect the runtime symptom to a specific operation.

Flow-Like's run inspection is the feature that most directly answers that night. A run leads from recorded evidence back to the relevant node on the Board. The application stops looking like one undifferentiated failure.

Large programs also need a dense way to work. FlowScript lets a developer search, compare, review, and edit the same logic as text. During an incident, the canvas offers a map; during a broad code change, the text editor may be faster. Both act on the same Flow.

I believe that structure would have shortened the original investigation. That claim is a counterfactual, not a benchmark. The defensible requirement is smaller: a failure should leave enough connected evidence for a capable responder to know where to begin.

1.4 Domain knowledge belongs in the program

The call exposed another distance. Domain experts understood the operation, its exceptions, its regulatory setting, and the cost of getting it wrong. Conventional delivery often passes that knowledge through several groups before it becomes code. The people who supplied the rule may then struggle to inspect its implementation.

Simpler visual tools help until the application outgrows them. Teams often respond by placing general-purpose code inside a workflow block. The canvas still shows a reassuring rectangle while the real behavior moves behind its label.

My background in game development suggested a stronger visual model. Unreal Engine Blueprints had shown that a graph could be a genuine programming surface and a practical route into software. Enterprise

workflows add another requirement: once the graph becomes large, experienced developers need text.

That combination led to the core design. A developer can use FlowScript for a large change. A domain expert can inspect the resulting process on the canvas. An operator can follow a failed run through the same nodes. Their expertise meets in one program.

The idea grew beyond the editor. Existing systems still have to connect to the logic. Applications need data, identity, permissions, versioning, deployment, and evidence. Flow-Like became the platform around those concerns; FlowScript became the text form of its Flow logic.

The standard remains concrete. When a Flow fails at three in the morning, a capable person who did not build it should be able to find a credible starting point before the system expert answers the phone.

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.

Flow-Like makes a trade. Executable operations enter a shared, typed model instead of arriving through arbitrary escape hatches. In return, the platform can inspect, run, and govern more of the application as one system.

The constraint has to earn its place. It should prevent a meaningful class of mistakes while leaving real work practical. Developers will route around a platform that makes the sound path needlessly painful.

The following principles are the standard against which FlowScript and Flow-Like should be judged.

2.1 Reliability begins during authoring

Teams often postpone operational work until a feature exists. Authentication, logging, recovery, permissions, secret handling, and cost controls enter separate backlogs. The prototype feels fast because it temporarily ignores the work required to keep it alive.

A reliable program has to survive conditions beyond its author's machine. Unexpected input should meet an explicit boundary. A failed dependency should leave evidence. Reviewers should be able to see what changed, and a responder should be able to connect a runtime error to the responsible operation.

Flow-Like therefore places typed connections, versioned changes, runtime context, and run evidence around the Flow. Deployment targets still differ in maturity and policy. The durable principle is that application

teams should spend their time on the domain problem instead of rebuilding a private operating model for every project.

Efficiency belongs here too. Runtime waste matters on a critical or high-volume path. Human waste appears when each team recreates the same infrastructure and rediscovers the same failure modes. The platform should reduce both.

2.2 Readability is operational

Software outlives the attention of its first author. Developers, domain experts, operators, and reviewers each arrive with different context. AI systems now join that readership and may also propose changes.

Inside Studio, the canvas exposes operations and relationships. FlowScript gives developers a compact text editor for the same logic. Both author the same Flow.

The two views answer different questions. Text works well for search, comparison, and broad edits. The canvas shows locality, paths, boundaries, and the node responsible for a run result. A readable Flow lets the team choose the useful view without maintaining two programs.

Organization affects correctness once several people have to review or operate the application. If important behavior disappears into a generic code box, the visible Flow can no longer support that work.

2.3 Hard work should move quickly

Safety becomes irrelevant when it makes useful software too slow to deliver. Flow-Like has to make the disciplined path competitive.

The node catalog turns recurring capabilities into typed operations. Inputs and outputs become pins; connections can be checked against their declared types; logs can refer to the exact node that ran. Platform services provide the surrounding application structure so each Flow can focus on its own decision or process.

This matters in an existing estate. A developer should be able to connect a current API, database, file store, or device and add only the logic that is new. A reusable package can expose a specialized integration once, with a contract that later authors can inspect.

The platform cannot decide the domain model or make an unreliable dependency reliable. It can remove engineering repetition that adds little value to the domain. The useful measure is the time required to reach software that a team can understand, extend, and operate.

2.4 Why text belongs beside the graph

Low-level building blocks keep a workflow honest. They also make the canvas larger. Related logic spreads across space, repeated patterns take time to place, and a long wire can be slower to follow than a compact expression.

A generic code node shrinks the visible graph by moving behavior into a second language. One box may conceal many calls, branches, and side effects. Debugging crosses into a separate toolchain, and the node's visible contract describes less of what actually happens.

FlowScript addresses scale while retaining the graph model. Its values, calls, branches, loops, functions, and handlers reconcile to operations the Board can represent. The language provides a purpose-built text form rather than a serialized configuration file.

The result is compact authoring that remains part of the graph model. If reconciliation cannot preserve that meaning, Apply must stop with a precise diagnostic.

2.5 Constraints must state their boundary

Typed pins catch incompatible connections before a run, within the facts declared by the nodes. They cannot prove that a business rule is sensible or that an external service will honor its schema.

Secret handling keeps protected values out of ordinary FlowScript source and guards sensitive changes. That protection is meaningful, although metadata alone cannot prevent a later node from logging or returning a

secret. Stronger claims require verification across previews, errors, storage, and deployment paths.

Custom nodes use WebAssembly components with declared host capabilities. Interactive clients can show package and permission information before execution, and publication can include review. Resource limits, remembered trust, and unattended runs still need explicit policy and testing.

Versions and publication gates follow the same rule. A numbered Board version freezes the graph; packages, data, credentials, and external systems can have separate lifetimes. Good documentation states what a control protects and where its responsibility ends.

2.6 Broad outcomes, opinionated implementation

Flow-Like aims to support applications, automations, agents, data work, APIs, and internal tools through a small number of visible execution concepts. That breadth depends on consistency at the implementation boundary.

When the catalog lacks a capability, an experienced developer can package a WASM node and expose a typed contract. A domain expert then uses the capability as a normal building block, without inheriting its implementation language.

AI follows the same path. An authoring agent proposes reviewable changes over declared operations. A runtime model call occupies an explicit node where semantic uncertainty is useful. Exact logic stays in deterministic nodes around it.

The platform deliberately limits arbitrary injection of credentials, deployment assumptions, side effects, and unreviewed code. The return on that constraint is collaborative software whose logic remains available after its first author leaves.

2.7 The product has to live by the manifesto

FlowScript and Flow-Like are evolving. Authoring parity has edges, clients differ, and deployment targets have different operating histories. Security, performance, scale, and cost claims need tests or measurements for the release being described.

This book marks changing capabilities as **Current**, **Preview**, or **Vision** and names known limitations near the feature they affect. A polished story is not worth a bad operating decision.

The same candor applies to product design. If the governed route stays slower or more awkward than the escape hatch, users will choose the escape hatch. The team building Flow-Like owns that problem. Constrained freedom works only when the reliable route is also a practical way to ship.

PART I · CHAPTER 03

One Platform, One Flow Model

Understand how Flow-Like Apps, Flows, Boards, integrations, Events, data, permissions, and execution fit together.

The introduction gave us the outline. This chapter fills in enough of the platform model to build and run the first application without stopping for a new noun on every page.

Flow-Like is the platform. Studio is its desktop application. An App groups one solution, and a Flow contains one executable process inside that App. The canvas and FlowScript editor show the same Flow logic in different forms.

3.1 App, Flow, and Board

An **App** is the project and governance boundary. It groups the Flows, Events, pages, files, structured data, packages, members, roles, and release settings that belong to one solution.

The word *App* describes ownership rather than interface shape. An App may deliver an automation, agent, API, data pipeline, analytical tool, or interactive page. Several of those surfaces can share the same logic and data.

A **Flow** is an executable process inside an App. Authors use the term when they say “run the triage Flow” or “publish version two of the ingestion Flow.” An App can contain several Flows with separate responsibilities.

The persisted graph behind a Flow is a **Board**. It stores nodes, pins, connections, variables, layers, comments, and execution settings. The canvas is the visual editor for that Board. FlowScript is its typed text form.

The practical chain is short:

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

3.2 Studio contains the authoring views

Studio is the complete desktop application. A developer uses it to manage Apps, edit Flows, inspect data, work with packages, run logic, and examine evidence. The node canvas is one part of that application.

The **canvas editor** shows a Board as nodes and typed wires. It is useful for tracing a value, following an execution branch, opening a layer, or finding the node that emitted a run result.

The **FlowScript editor** renders the same logic as declarations, calls, expressions, branches, loops, functions, and Event entries. Text is efficient for search, review, and large changes.

In the current implementation, Studio persists the Board. Opening the FlowScript editor renders that Board as source. Applying an edit parses the source and reconciles it with the existing graph. The Rust runtime then executes the updated Board.

Node identity matters across the round trip. An anchored statement can update the same node instead of creating a replacement, which preserves details that source does not need to repeat. Later chapters examine anchors and reconciliation. For now, the working rule is enough: choose the editor that fits the task; both change one Flow.

3.3 What a Flow contains

A **node** is a typed operation. It might transform text, call a service, read a variable, write a file, branch execution, or mark an entry into the Flow. A node has a catalog identity, documentation, declared inputs and outputs, and execution behavior.

Nodes communicate through **pins**. Data pins carry values with declared types and shapes. Execution pins carry control order. A **wire** connects compatible pins.

These relationships answer separate questions. A data wire tells us which incident record reaches a storage node. An execution wire tells us when that write may happen. A value can be available before the application is allowed to produce its side effect.

A **pure node** has no execution pins and is evaluated when another operation needs its value. An **impure node** joins the execution path because its order matters. String conversion can often be demand-driven; sending a message or updating state needs explicit control.

Layers organize a larger Board. They let a group of nodes sit behind a named, typed boundary while its implementation remains inspectable. Function layers also give FlowScript callable functions. A layer reduces visual noise without turning its contents into arbitrary hidden code.

FlowScript compresses these graph relationships into familiar syntax. An expression may lower to a pure node, a statement may add an impure node to an execution chain, and a branch becomes visible control structure in both editors.

3.4 Existing systems stay in the picture

Few applications begin with empty infrastructure. Flow-Like is designed to orchestrate and extend systems that already hold the company's data and business rules.

Catalog nodes can call external services, work with files and structured data, and use configured credentials. Packages add integrations or domain-specific operations behind typed contracts. A developer can wrap an existing API once, then let later Flow authors use it as a normal node.

The connected system can remain the system of record. A Flow may validate input, coordinate calls, add a new decision, or expose a safer interface while the existing service continues to own its data. This allows a team to improve one process without funding a full migration first.

Integration stays visible on the Board. Reviewers can see where data leaves the App, which node needs a credential, and how the Flow handles the result. Runtime secrets and provider credentials remain separate from ordinary FlowScript source.

This brownfield path is central to the platform. New Apps can grow around the estate, and useful pieces can move into shared packages or Flows as their ownership becomes clear.

3.5 Events connect a Flow to callers

Flow-Like uses *Event* for an entry inside a Flow and for the configured exposure around it. The two objects serve different jobs.

An **event node** begins execution on the Board. Its output pins define the values available to the Flow. The node's kind also limits which external surfaces can expose it.

An **App Event** selects that entry and configures how a caller reaches it. Depending on the entry and environment, the surface may be an API, schedule, quick action, chat, page, or another supported integration. It also chooses a Flow version and an execution location where those options apply.

This separation keeps business logic independent from one transport. A typed incident handler can serve a form and an API through thin adapters. A production Event can stay pinned to an immutable Flow version while development continues on the latest draft.

When this book says *event node*, look inside the Flow. *App Event* names the configured bridge to a caller.

3.6 Data, people, and execution share the App boundary

An App can own files and structured data that its Flows read or write. Data Studio provides the workspace for tables, SQL, reusable queries where supported, and domain models such as ontologies. External stores still keep their own operating properties; App ownership describes the logical relationship to the solution.

The App also carries members, roles, publication state, and Flow access. Several security boundaries meet here. Platform identity establishes who is signed in. App membership governs work inside the project. Event authentication covers exposed entry points. Node capabilities and runtime credentials determine what an operation can request and reach during execution.

Compatible Flows can run locally through Studio or remotely through configured Flow-Like services. A Hybrid Flow may run in either suitable environment as a whole; it is not divided across a laptop and server during one execution. A configured App Event selects Local or Remote because its caller needs a definite destination.

The runtime loads the chosen Board and version, resolves its node catalog and runtime context, and follows data dependencies and execution wires. A run can emit logs, progress, state changes, interface messages, results, and stored artifacts. Those records give Studio evidence to place back on the Flow.

Local and remote environments have different capabilities, trust boundaries, and operating histories. Later chapters state release-specific guarantees. The stable mental model is the shared Board and execution contract.

We now have enough vocabulary to build **Incident Triage**. It will be one App with one Flow. An event node will start the Board, typed nodes will transform and classify a report, and the canvas and FlowScript editor will show the same logic. Chapter 4 turns that model into a running program.

PART I · CHAPTER 04

First Flow: Incident Triage in Two Views

Build a deterministic incident triage Flow, edit its Board through FlowScript and the canvas, inspect failures, and save a tested version.

Our first Flow accepts an incident report, normalizes the text, classifies it with a deterministic rule, and records the decision at the appropriate log level. It uses no external services, so any change in behavior can be traced to the input or one of six visible nodes.

We will edit the same Flow on the canvas and in FlowScript, inspect a run, and save 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 match the generated node declarations and reconciliation tests. Exact canvas gestures, an end-to-end run, screenshots, and the malformed-input trace still need verification against the release used for publication.

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.

Here, “respond” means writing to the run log. API responses, notifications, and incident storage can wait until we understand the two authoring views.

The literal phrase check is only a teaching rule. Given the same string, it always takes the same branch, which makes the Flow useful as a 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}`

The normal report follows the Info path. The urgent report loses its surrounding spaces, matches despite its capitalization, and follows the Error path.

The malformed case depends on the release and invocation surface. A typed form may block the submission, an API adapter may reject it before the Board starts, or a string node may reject it during the run. We will record the observed boundary.

The `log::error` operation records an Error-level message and completes successfully. Some run lists may still color or classify the run from its highest log severity. Check the node evidence before treating an urgent classification as a runtime failure.

4.2 Build it visually

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

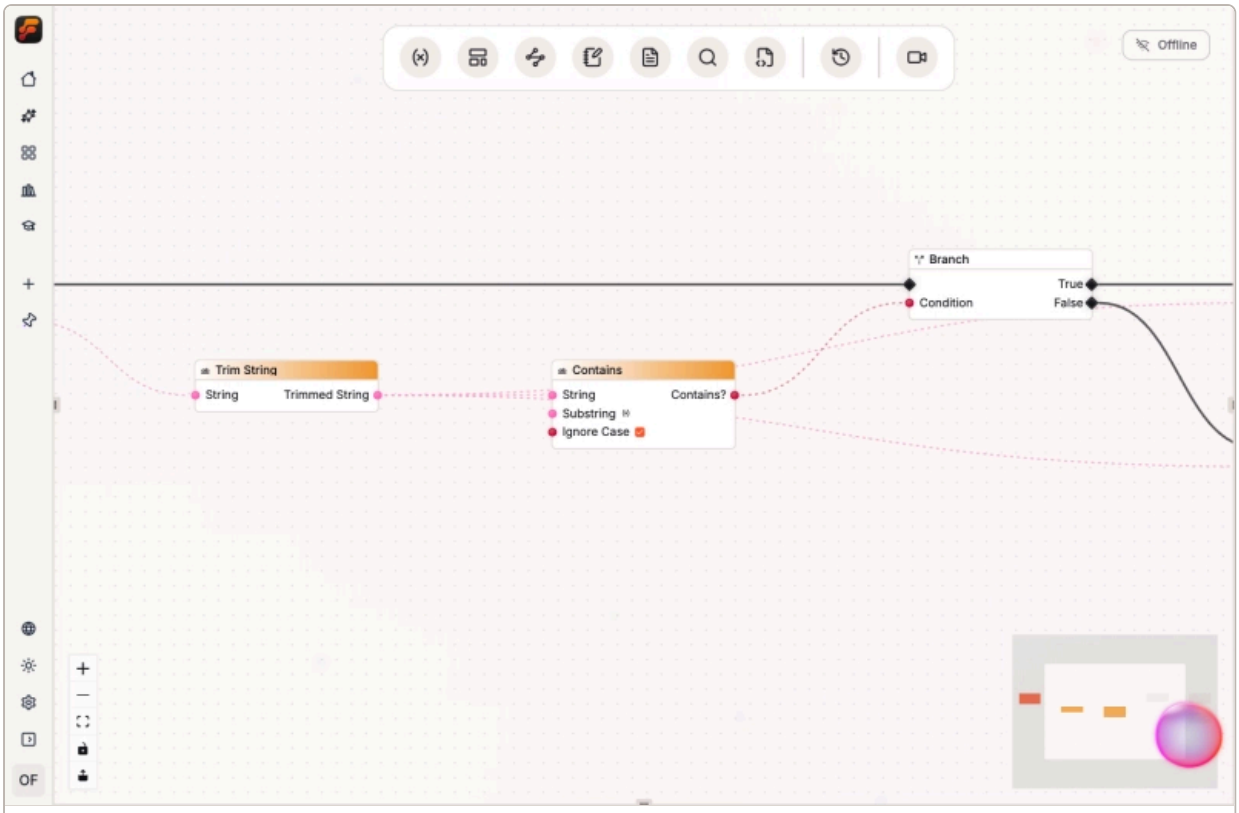
NODE	CATALOG IDENTITY	RESPONSIBILITY
Generic Event	`events_generic`	Starts the Flow and exposes `report: string`
Trim String	`string_trim`	Removes leading and trailing whitespace
Contains	`string_contains`	Checks the normalized report for the incident phrase
Branch	`control_branch`	Selects the urgent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

Displayed names may change. Catalog identities are the version-matched 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 path is now visible on the canvas:

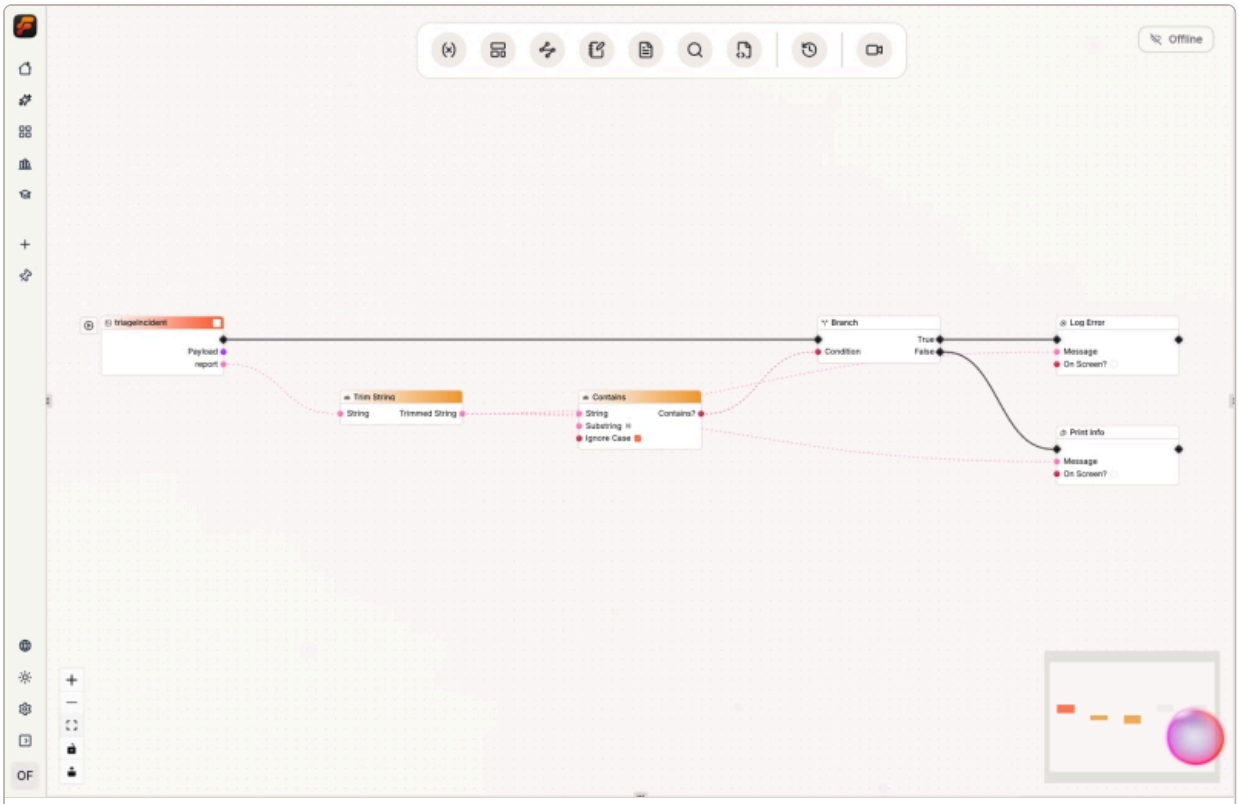


A focused crop of the first data path after FlowScript reconciliation and balanced auto-layout. Open the image to inspect the pins at full size.

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.

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 Flow has two kinds of wires:



The complete six-node Flow on the Studio canvas. Solid dark wires carry execution; colored dotted wires carry typed data.

Trim String and Contains need no execution wires. They are pure operations, evaluated when Branch needs its condition. Branch and the log nodes are impure because their place in the execution path matters. Data wires answer “Where does this value come from?” Execution wires answer “When does this work happen?”

Before opening the text view, read the Flow from left to right:

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

If the canvas does not communicate that sentence, improve the layout before moving on.

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 })
    }
}
```

FlowScript is the typed text form of the Flow logic shown on the Board. This source and the canvas describe the same six nodes.

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 several static calls from `log`. Fully qualified calls such as `log::error(...)` remain valid.

The event declaration describes the entry node:

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

`eventsGeneric` identifies the event kind. `triageIncident` names this entry. `payload: Struct` is built in; `report: string` is the typed output we added.

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

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

`report.trim()` represents the visible Trim String node. For nodes with a receiver pin, the value left of the dot supplies that input. The default output becomes `normalized`.

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 substring and comparison option. Its Boolean output feeds Branch.

The `if` block renders the Branch node as control flow. Its first block is the True output and `else` is False.

Each statement inside those blocks is an impure logging node:

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

The import makes these aliases available without a prefix. In a fully qualified call, `::` separates the namespace from the node alias. A dot accesses a field or receiver operation. The object keys are input pins.

The editable view may include identity anchors such as `//@n:...`. They associate a statement with an existing Board node. The book hides them for readability; preserve them while editing unless 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"
```

Before applying, inspect the reconciliation preview. It checks the source against the current Board and plans the change without mutating the saved Flow.

The exact number of internal commands may change between releases. Check that the preview updates the existing node instead of replacing the Flow or all six nodes. If it does more, inspect the source and its anchors.

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

The edit is parsed and reconciled into the existing Board.

Use source-to-node navigation if the release provides it. Publication verification should capture the pending plan and changed node.

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.

The canvas edit changed the Board; newly rendered FlowScript now describes that change.

When the Board changes under a dirty FlowScript buffer, Studio blocks Apply until you refresh or merge the stale buffer and resolve every three-way merge conflict. Render fresh FlowScript after a canvas change.

Remove the temporary warning and confirm that the source has returned to the six-node baseline.

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.

The Log Error node still completed successfully. Some run lists use the highest log level for color or classification, while remote execution stores completion status and log severity separately. Read the node-attributed evidence before deciding whether the run failed.

Now try the malformed payload:

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

Record what the chosen release actually does.

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 assume Trim String failed because it is the first string node. The platform may reject the wrong type earlier.

When a true node failure is available, open its run and navigate from the relevant log entry to the responsible node. Capture the input, Flow version, run identifier, log level, message, and focused node. That record gives the next responder a concrete place to start, which matters far more on a Board with hundreds of nodes.

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 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.

The saved version now contains one tested Flow whose canvas and FlowScript views describe the same logic. Chapter 5 explains how its data and execution wires behave at runtime.

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 Incident Triage Flow carries data from Trim String through Contains into Branch. A separate execution path starts at the event and reaches one logging node. One path supplies the decision; the other decides what runs next.

This separation underpins Flow-Like's execution model. FlowScript expressions describe value-producing nodes, while statements and control blocks describe execution wiring. When a Flow surprises us, trace execution forward and data backward.

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 the release used for publication. Configured defaults persist; runtime pin values are deliberately transient.

5.1 A node is a typed operation

A node is the smallest named operation shared by Flow-Like's authoring tools and runtime.

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 spelling still resolves to a catalog node. Open the Flow in Studio and the canvas shows the operation, its pins, and its place in the Flow.

Its catalog identity carries a contract. 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.

Nodes may omit some metadata, and metadata cannot prove an implementation correct. It still gives the platform a surface to inspect before execution and an identity for later run evidence. Custom WASM nodes, for example, request specific capabilities such as networking, storage, model access, or function calls. Later chapters examine how each execution mode enforces them.

A domain expert can point to a business step by name. Developers see the same identity in its contract, and operators see it again in the run log.

Two terms need precise definitions. *Standard* refers to catalog classification, not whether a node is built in. A *pure* node has no Execution pins. A node with any Execution pin is impure, regardless of its size or source package.

5.2 Data pins carry values

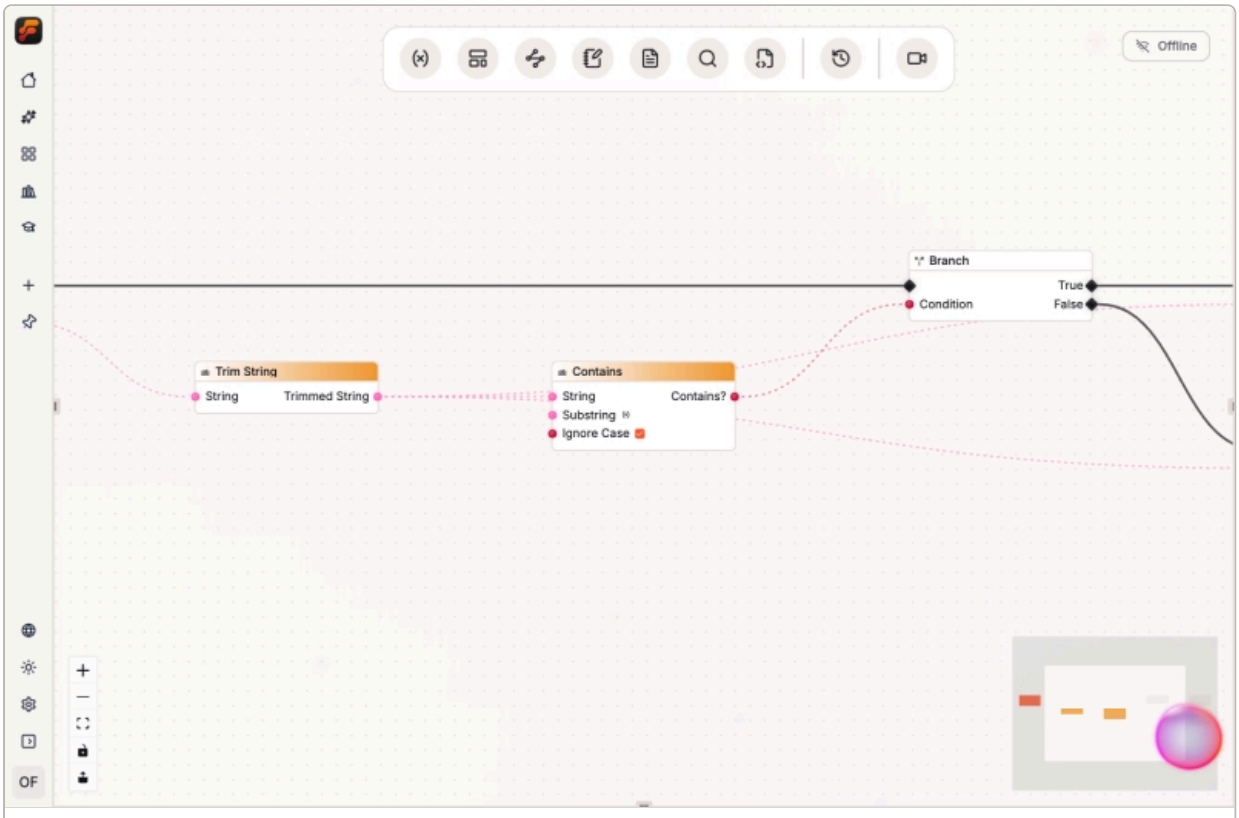
Pins are a node's public boundary. An input pin states what the operation accepts. An output pin states what it produces. A data wire connects an output to a compatible input and answers one question:

> *Where will this input get its value?*

Direction is part of the contract. Two pins labeled `Message` are incompatible if both accept or both produce a value. Studio rejects those connections, along with self-connections.

The graph model also distinguishes strings, integers, floats, booleans, dates, paths, bytes, structured values, generic values, and Execution. Value shape separates scalars from arrays, maps, and sets. Structured pins can carry a schema with known fields.

Our first Flow already used this system:



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

Contains can feed Branch because both pins use one Boolean. Wiring that output to a String input fails during authoring, exposing the type error before a production run.

Generic pins provide controlled flexibility. A generic formatter may accept several concrete types. Once a wire resolves that generic, connected pins must remain compatible with the resolved contract. Type inference still enforces the resulting type.

Pins may also define choices, numeric ranges, sensitive literals, or schema rules. Studio uses this metadata to build the appropriate editor.

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.

Execution can therefore reach a node with no wire on one input. The node uses its default if the contract provides one; otherwise evaluation may fail there. Data wires do not schedule nodes.

The Board persists definitions, connections, and configured defaults. Values evaluated during a run live in memory and are excluded from pin serialization. A node may log selected evidence, but persisting every value would be costly and could copy sensitive data into logs. Evidence belongs to the node contract and the organization's policy.

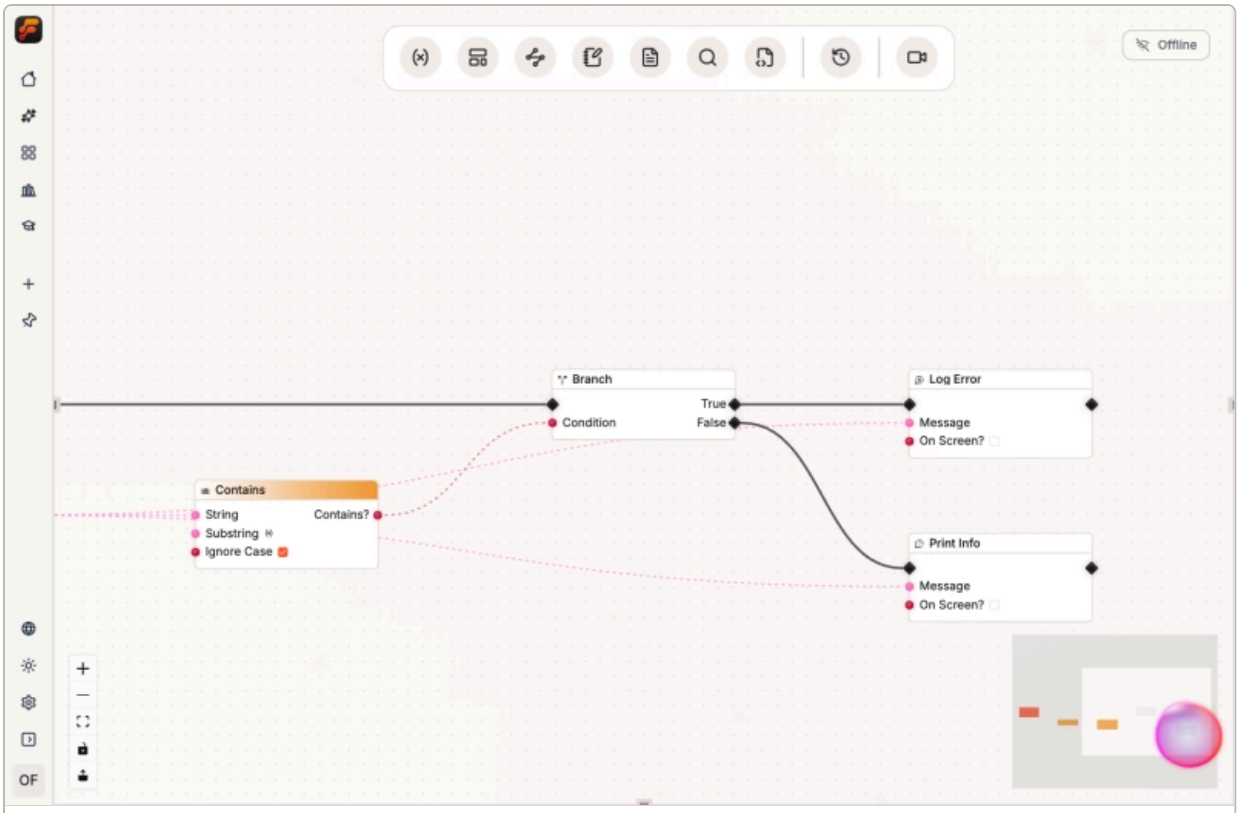
For debugging, the practical rule is to work backward. Begin at a surprising input and follow its data wire to the producer. Repeat until the first surprising value or missing source appears. Looking only at the execution path cannot answer where the value came from.

5.3 Execution pins carry order

Execution pins determine when control may proceed. Application data travels on data pins.

An execution input asks what must activate before the node runs. An execution output selects the path eligible after completion or an outcome. The Studio canvas currently draws these as diamond pins. Color and styling may change; the Execution type defines their behavior.

Return to the Incident Triage Flow:



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

The event starts control. Branch evaluates its Boolean input and activates one execution output. The selected logger runs when control reaches it; a populated Message input alone cannot start it.

FlowScript renders this part of the Flow 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` represents the Branch node and its True and False execution outputs. Statements in each block are the impure nodes wired to that output.

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 forward path. Its ordinary successors are not queued, and the run records the failed node. Recoverable failure stays visible on the Board.

Some nodes define outcome paths. The current API Call node exposes Success and Error outputs. A completed HTTP exchange selects one based on status and keeps the response as data. A transport failure produces a node error instead of a completed response outcome.

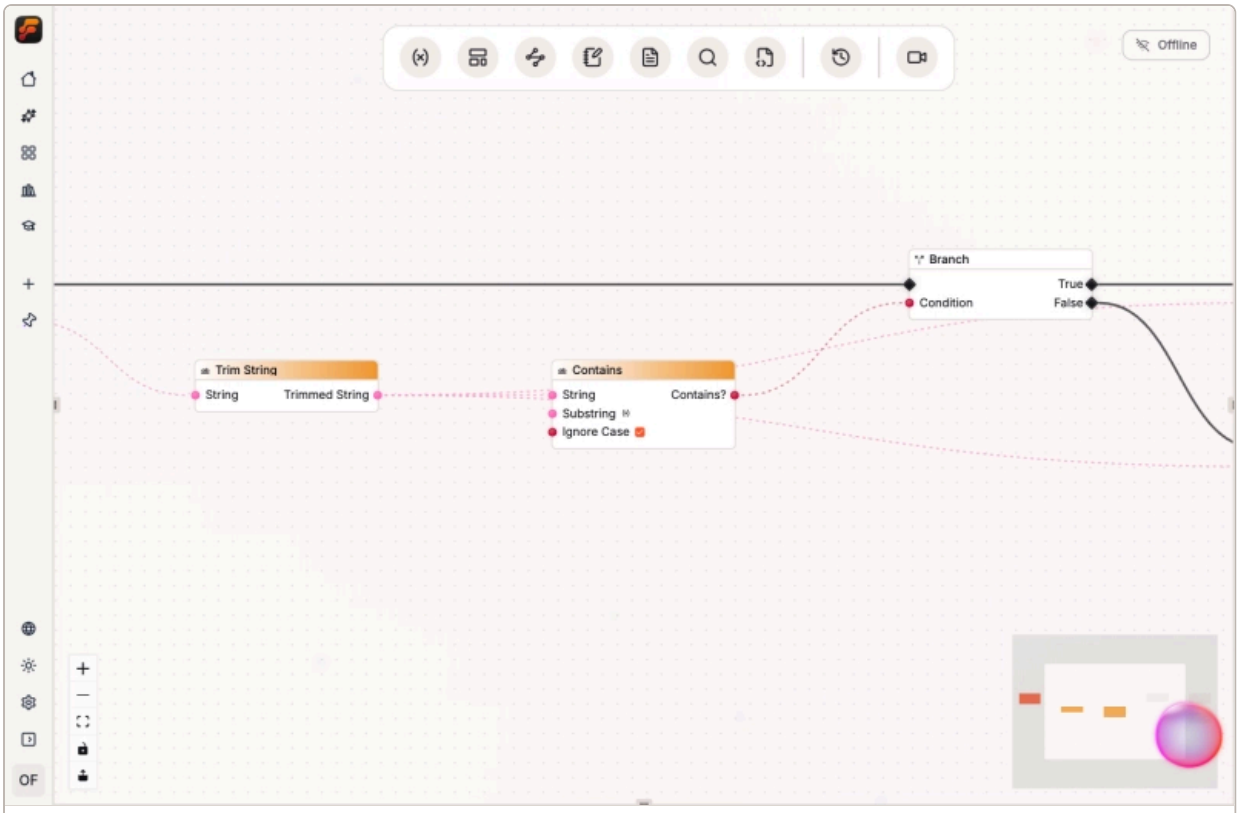
For an unexpected error on any executable node, including one with a normal Error outcome, the current Studio toolbar can enable **Handle Errors**. It adds an **On Error** execution output and an Error string output. A connected handler runs after failure. The original node stays in Error state and retains its attributed evidence, even when handling prevents the error from unwinding the run.

The Flow author can route expected failures into a retry, fallback, ticket, or human review. The organization's policy determines whether the Flow recovers or stops.

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 to the consumer without execution wires. An unconsumed pure chain does not run merely because an event started. Producers do not fire their consumers; executing nodes pull the values they need. A pure node may be evaluated zero, one, or several times depending on its consumers and execution contexts. Work that must happen once, costs money, or changes an external system belongs on an explicit execution path.

The runtime detects purity structurally: any Execution pin makes a node impure. Node authors must declare that boundary honestly. Giving a network call only data pins would conceal cost and failure behind demand-driven evaluation. Declared WASM capabilities add enforcement, though pin shape alone cannot prove 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, use two passes:

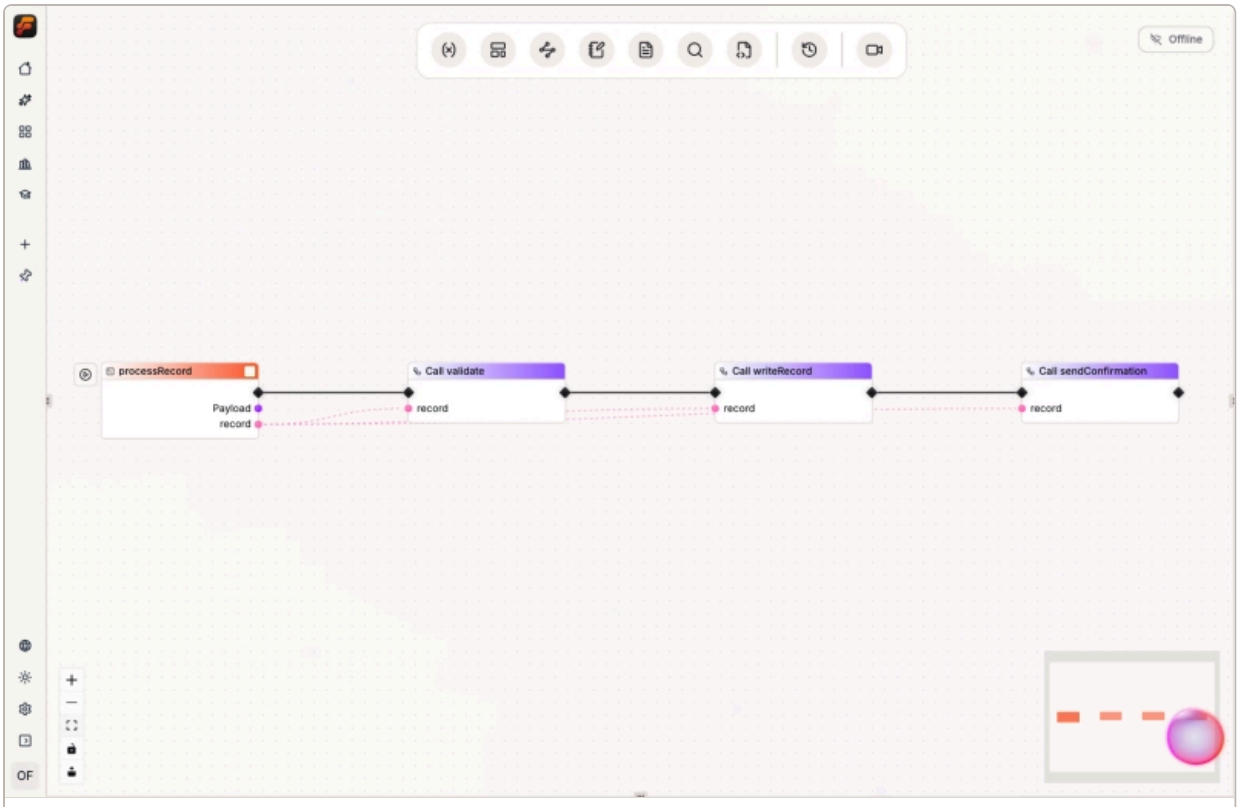
1. Follow execution forward from the event and ask whether control could reach the node.
2. Follow data backward from the surprising input and ask whether its value could be produced.

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

5.5 Explicit sequence and explicit parallelism

Canvas position helps readers but cannot schedule work. Execution pins and control nodes define order and concurrency.

For ordinary sequential work, draw one unbroken execution chain:



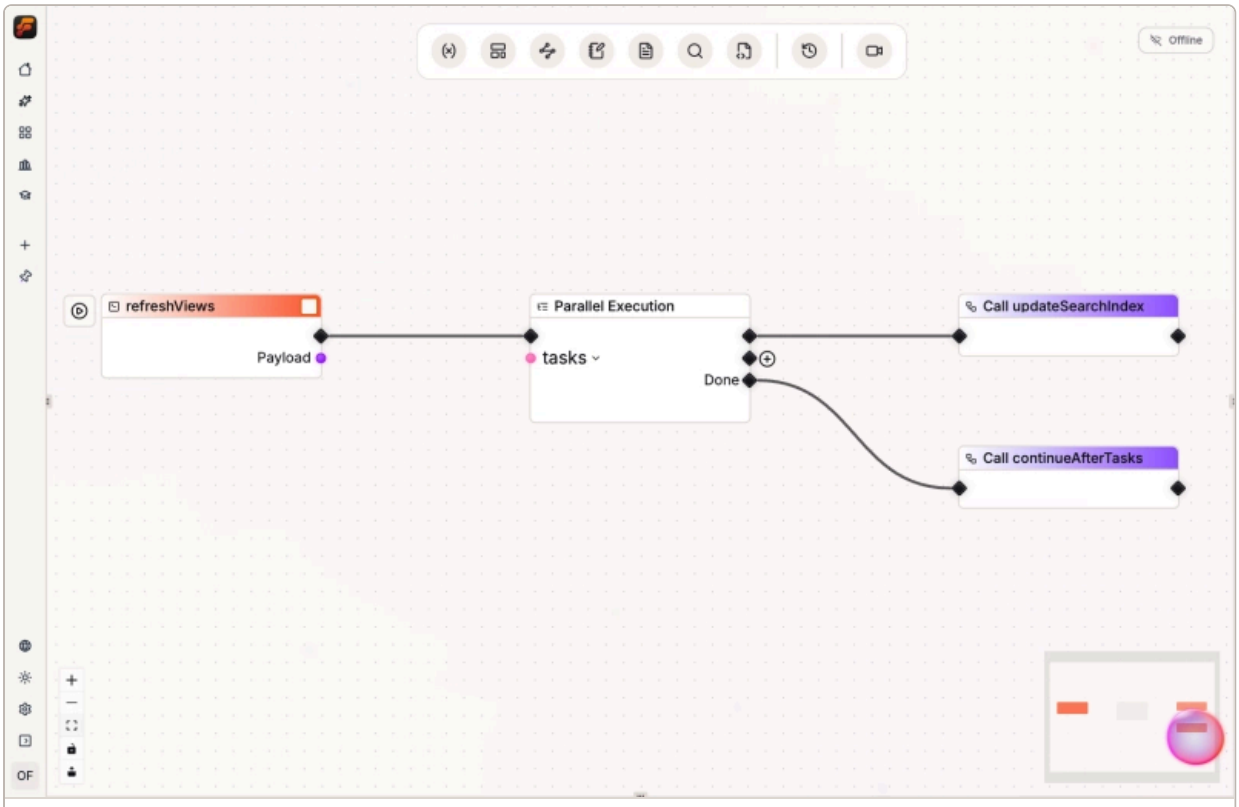
Consecutive impure FlowScript calls reconcile into one explicit execution chain.

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

For several ordered branches, use a Sequence control node. Its outputs own 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`.

Task paths do not wire back into Done. Parallel activates that continuation after its branches settle. This example wires one task arm and Done; Studio can add 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.

Branches may finish in any order. Avoid races on shared mutable state, and ensure external effects can tolerate that ordering.

Ordinary topology fan-out isolates ready siblings. One failed branch stops its successors while other ready targets continue, bounded by executor capacity. After the siblings settle, an unhandled failure makes the run's terminal status **Failed**. A failure that reaches and completes an explicit handler remains visible on its node and in attributed logs without failing the run.

The dedicated Parallel Execution and Sequence nodes currently differ from that general rule. They can record a child error without propagating it to the run's final status, and Parallel Execution can still proceed to Done after collecting its children. Inspect child logs instead of relying on the outer control node. Test this behavior against the release used by the book.

Sequential paths behave differently, and purpose-built nodes may define their own outcomes. Add an explicit join when all work must finish. Model cancellation or compensation explicitly and verify the selected nodes in the target release.

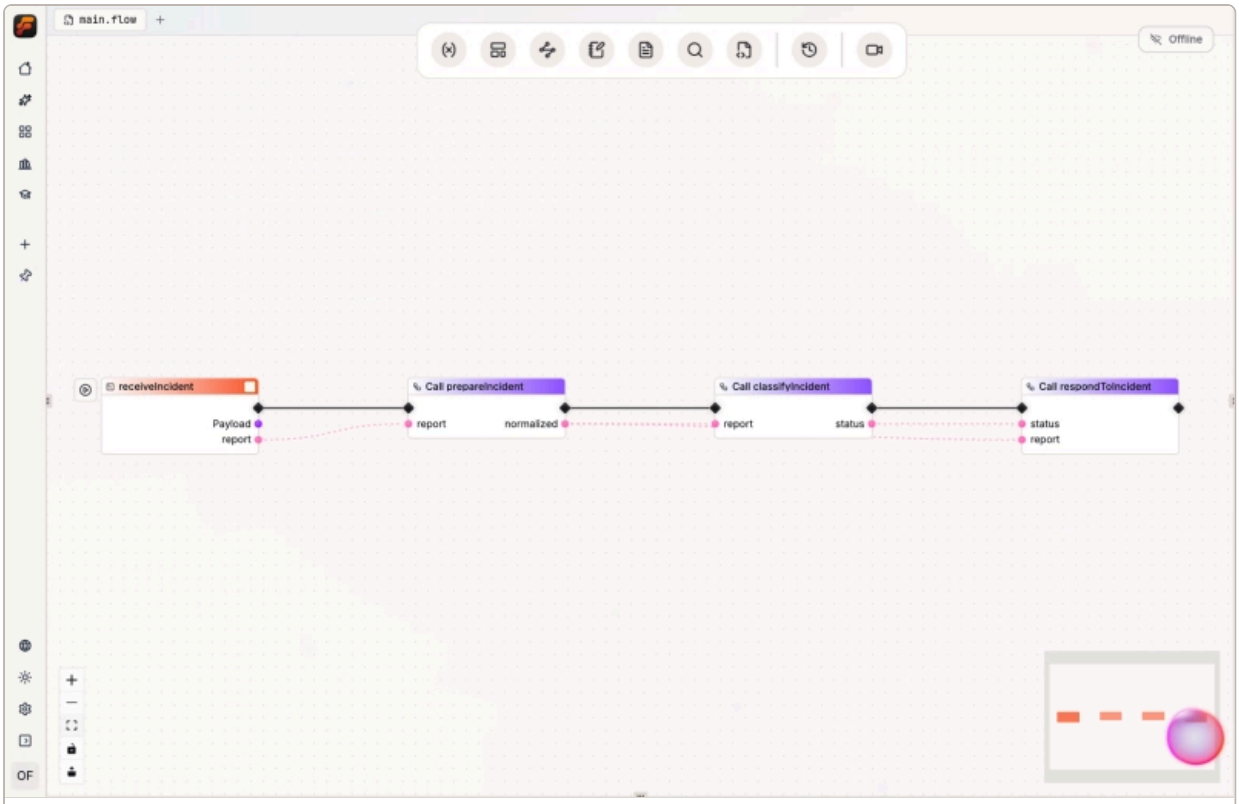
During an incident, the pins show what was ordered, what ran concurrently, where branches joined, and how failure was routed.

5.6 Layers organize; functions are callable

Large Flows need a way to hide detail without hiding behavior.

A layer folds nodes behind a named, typed boundary. Crossing wires become boundary pins. Opening the layer reveals the same nodes, dependencies, paths, and comments. Collapsing it changes only the view; execution remains the same.

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 refers to a function layer. Opening `prepareIncident` reveals its maintained implementation and signature. Collapsing the same nodes into an ordinary **Prepare Incident** layer would organize one inline region without making it reusable. Start and Return boundaries show the values and execution paths that cross the edge. Inner failures remain attributed to their nodes.

A plain layer organizes one inline Board region. A function layer defines reusable logic. Its boundary pins form a signature, and each invocation appears as a Call Function node. FlowScript renders that Flow logic with function syntax.

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.

Name layers and functions after their responsibility, keep boundaries narrow, and review boundary changes like API changes. The next chapter shows how FlowScript records these structures in one document.

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 the typed text form of one Board, arranged for stable reading rather than authoring history.

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 emitted for a chain with no reachable entry. Its contents never run, so it usually signals a Board problem rather than something to author deliberately.

This chapter maps each document section to its Board entity and clarifies Flow-specific meanings behind familiar syntax.

Release check: The complete Incident source is a parser-tested canonical fixture at `examples/document-anatomy/anatomy.flow`. Catalog-aware reconciliation, execution, and editor diagnostics still need verification in the publication release. Free-standing `//` comments in bodies survive parse-and-format, but the Board reconciler does not persist them as canvas comments. 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.

The resemblance to TypeScript and Rust helps with reading, but the Board model supplies the semantics. FlowScript runs no hidden JavaScript engine, and decorators are fixed Flow metadata rather than 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 })
}
```

Stable rendering keeps diffs meaningful, reduces equivalent spellings for generated edits, and lets repeated parsing and rendering converge without whitespace churn.

Section order may therefore change after formatting. The renderer groups top-level declarations by role even if the input places an event before a function. Board lowering sorts variables by name and stable identity; function layers and event entries use deterministic identity and entry ordering. Runtime order lives inside bodies, never in top-level placement.

Formatting parses and renders syntax without consulting the node catalog, so it can format an unknown call. Apply resolves declarations, checks pins and types, and plans the Board changes.

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 parser and formatter preserve a free-standing body comment and may move a trailing comment to its own line. The reconciler currently drops these comments when writing to the Board, while the text AST skips top-level comments. This remains a round-trip gap.

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 an existing entity by identity. The authoring surface emits them when requested; book examples normally hide them.

Edit anchors carefully. Preflight checks whether one anchor is assigned to two distinct entities before Board lookup and produces a diagnostic, including when the ID is absent from this Board. A multi-output Event header and its immediate first arm-routing Branch may repeat one node anchor because they represent the same Event entry. After that check, an ordinary node, variable, Function, or module anchor whose ID is absent from the current Board is reported as a correction and treated as unanchored. The normal resolver can then create the entity or reuse one unique compatible target. Absent Event anchors keep their specialized recovery path. It re-anchors one compatible entry or creates a fresh entry instead of guessing between zero or several candidates. An anchor that resolves to incompatible live Board state, or an ordinary unanchored declaration with several possible targets, fails closed. Removing anchored sections can trigger a deletion plan that requires explicit approval.

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. A dot handles field or receiver access 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 })
```

Current lowering can derive a glob import when a Board has at least two static calls in one namespace and no name collision. That is why Chapter 4 renders `use log::*` above `error(...)` and `info(...)`. A single or ambiguous call usually stays qualified.

Unknown namespaces, missing members, reserved aliases, and name collisions produce reconciliation diagnostics. Argument shape may disambiguate the same member exposed by two globs; otherwise the call needs a qualified spelling. An unused valid import is a non-blocking correction.

`use` is top-level only and changes name resolution in this document. Package installation, capabilities, and App approval remain separate.

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, accepts a string or `null`, and defaults to `null`. `tags` is an optional string array with an empty default. Interfaces also support named types, maps, unions, literal string alternatives, `any`, and quoted field names for invalid identifiers. Chapter 7 covers their type behavior.

A bare `Struct` says little about its fields. A named interface gives a top-level variable, Function boundary, or Generic Event output an exact 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.

`interface Incident` describes a pin shape. It creates no runtime class, constructor, prototype, or methods. Method-shaped calls still resolve to a catalog node or declared FlowScript function with a compatible receiver contract.

`@schema("…")` remains a legacy escape for metadata a named interface cannot represent. Prefer an interface when it preserves the schema; avoid pasting large JSON Schema into ordinary source.

Interfaces do not support decorators. They are derived from schema-bearing text surfaces rather than stored as independent Board assets, so a declaration needs a variable, function, or event boundary to describe.

6.4 Top-level variables

Top-level variable keywords have Flow-specific meanings.

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

Both keywords allow assignment during a run. Each run receives a fresh in-memory value from a permitted runtime or Event override, or from the persisted Board default. Changes disappear after the run; durable 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

DECORATOR	MEANING
`@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.

`@readonly` locks ordinary edits to the variable definition and configuration; it does not yet guard runtime writes. `@runtime` selects a runtime-configuration channel. Current Web and Desktop storage is local to the client profile and device, while an Event can carry a trusted override. `@secret` still forbids credentials in source. Chapter 11 covers the enforcement boundaries.

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 the declaration. Reconciliation rejects a non-empty secret initializer without echoing it in the diagnostic. Credentials enter through a trusted secret-setting surface, and existing defaults are omitted when the Board becomes text.

Local bindings are separate. Inside a body, `const normalized = report.trim()` names a node output. A local `let` alias or typed variable does not affect Board-variable exposure.

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 an input pin on the Function layer and the named return becomes an output pin. The body remains visible graph logic inside that layer; each call becomes a Call Function node targeting the layer:

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

The declaration's return syntax is deliberately explicit:

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

Parentheses hold a named output list rather than a tuple. Each output name belongs to the pin contract, and returned values must match the declared count and types. Mismatches produce a diagnostic.

Purity determines the execution boundary. The function above contains demanded data work and needs no Execution pins. A function with an impure call gains one execution entry and continuation; each call joins the caller's path. It uses the Chapter 5 runtime model.

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

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

This `const` is an SSA-like name for a node output. When a local binding merely aliases a non-call expression, canonical rendering uses `let`. Top-level exposure rules do not apply here.

Functions may carry `@cache` metadata. The bare form uses the default namespace, a five-minute lifetime, and App scope. A structured form sets namespace, TTL, and App or user scope. A cache hit skips the body, including side effects, so use it only when inputs determine the output.

Function anchors use `//@l:` because the layer holds the durable identity. Inner nodes use `//@n:`. Changing the name of an anchored Function emits `RenameLayer` for the same Function layer ID, so its body and existing call targets keep their identity. The new name must remain unique in its module. Changes to an established Function's parameters or returns are still rejected because the reconciler cannot migrate its boundary and callers safely. Copying without the anchor creates a new plan.

6.6 Events

Events turn the preceding declarations into runnable entries.

An event header has two names with different jobs:

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

`eventsGeneric` selects the catalog event type. The optional `triageIncident` names this entry. Without it, `eventsGeneric(...)` keeps the catalog default.

Parameters become the event node's data output pins. Generic Event can declare custom outputs; fixed event kinds use their catalog contract. On an anchored event, changes to parameter name, order, type, container, or schema are rejected as boundary-contract changes.

An event block is the entry node inside a Flow. An App Event is the platform surface that exposes it as a form, API, schedule, quick action, or another 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 from top to bottom: the import opens `log`, the interface defines the event schema, and the two Board variables hold internal and exposed configuration. The function creates a reusable pure layer. The event supplies its outputs, calls the function, and selects an execution path.

Before the Board changes, the document passes phased checks. Syntax errors carry a one-based line and column. Reconciliation resolves declarations, validates pins, types, schemas, returns, event boundaries, and execution wiring, then enforces structural limits.

The backend classifies findings with stable `FS_*` codes and phases such as parse, catalog resolution, type checking, execution wiring, lowering, and validation. Reconciliation source spans are best effort because the structured sidecar derives from a string diagnostic channel; repeated calls may yield several candidate spans. Read the code, message, expected and actual values, declaration or pin, and repair guidance together.

The server applies a FlowScript document as one program. Any reconciliation diagnostic blocks all planned Board commands. Non-blocking normalizations, such as an unused import, return separately as corrections. Deleting existing Board entities still requires approval, even with a clean plan.

Chapter 7 follows the types and schemas carried by these declarations onto visible pins.

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 describe what may cross a visible connection. A `string` maps to a compatible string pin. An `Incident` gives a Struct pin a visible field contract. An array carries its collection shape as part of the wire.

The governing rule is simple:

> *Reject known incompatibilities during authoring. When reality violates an unknown assumption, fail at the first node that discovers it and make that node easy to find.*

External contracts drift. Flow-Like uses the facts it has and makes the remaining uncertainty visible at runtime.

Release check: The scalar and container spellings, interface grammar, schema projection, connection diagnostics, and Make/Break behavior match the current parser, renderer, reconciler, pin-matching code, and catalog tests. Verify the complete examples and diagnostic navigation in the publication release. The final section records current schema-projection limits.

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 typed control connections. They answer “what runs next?” rather than “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 come from catalog nodes or typed boundaries. A quoted timestamp remains a `string`; use an explicit conversion before date operations.

`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 and may omit field information. That suits open boundaries such as webhook payloads, but gives Flow-Like less information than a named interface.

`any` is the textual spelling of a Generic pin. Its base type may specialize when connected to a concrete type. Generic array utilities, selectors, and external boundaries sometimes need this.

Use `any` where uncertainty is real. A concrete type improves catalog search, rejects incompatible wires, and exposes fields. An `any` value often needs inspection or conversion at runtime.

A schema-aware Struct can use **Make Struct (Schema)** and **Break Struct** with visible field pins. An open Struct relies on generic Get Field and Set Field operations with string keys, which are harder to inspect and govern.

7.2 Value shapes

Flow-Like records both the base type and value shape:

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

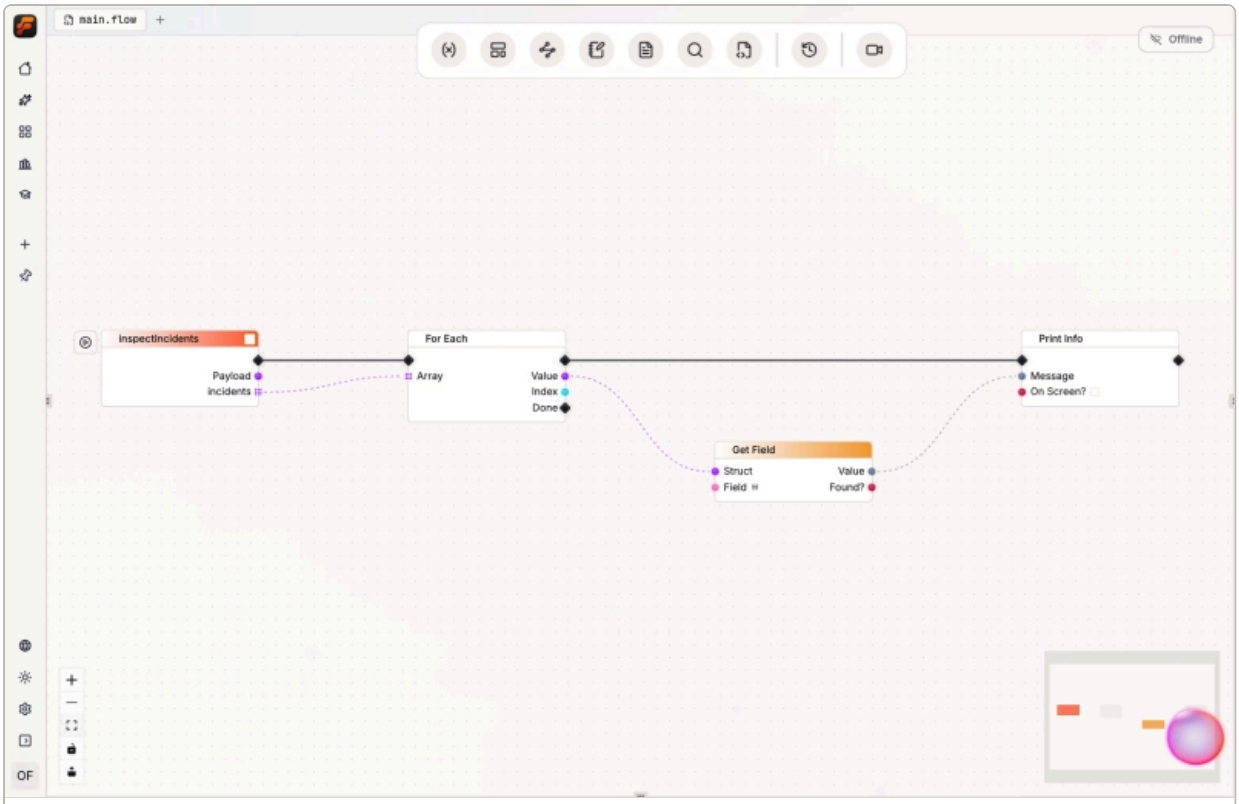
FlowScript renders those shapes in familiar syntax. Top-level Board variables may intentionally have no persisted default, as in this type-focused example:

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

Maps use `string` keys and one value type. Sets have one element type. Ordinary declaration and function-boundary types currently wrap one base type in one collection shape.

Shape affects compatibility. `string`, `string[]`, `Struct`, and `Struct[]` are distinct contracts. A Generic pin may specialize its base type, but connection planning still protects its declared scalar or collection shape.

A node producing `Incident[]` needs a selector or iterator before a consumer expecting one `Incident`. This example uses For Each:



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

Studio rejects a direct connection because `Incident[]` and `Incident` have different shapes. A selector or iterator must choose an element first.

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 promises only an array. The second keeps the element type visible. Empty arrays also need an annotation when their element type matters.

An object literal creates a `Struct`. Struct fields may have different types; Map values share one type under arbitrary string keys. The distinction decides between named field pins and key lookup.

FlowScript has array and JSON object literals. Maps and sets are normally built through catalog nodes, so construction remains visible on the Board.

7.3 Inference and its boundaries

FlowScript infers a top-level variable type only from an unambiguous literal:

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 scalar annotation when the literal already proves the same type:

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

becomes the canonical form:

```
const retries = 3
```

It keeps an annotation that adds information. These declarations have different types:

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

The first is an integer; the second is a float with a whole-number default. Structured and array defaults keep annotations so their intended shapes remain visible.

`null` carries no future value type, so FlowScript rejects this declaration:

```
const owner = null
```

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

```
const owner: string
```

This declares an unset default. It does not make `null` valid on every string pin. Interface fields can spell nullability as `string | null`; ordinary variable and boundary types currently use the simpler base-plus-container form.

Top-level inference stops at literals. This call cannot initialize persisted configuration:

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

A top-level Board variable stores configuration. Node calls belong inside functions and events, where catalog declarations and pins supply output types.

Studio checks candidate wires, and FlowScript reconciliation resolves calls against the live catalog. A known `int` to `string` mismatch, collection-shape conflict, or enforced-schema conflict blocks the new edge with a conversion diagnostic. Apply does not yet prove every direct literal against its target contract.

Execution catches unknown changes. If an external service starts returning a numeric customer ID without updating its contract, the first typed node unable to consume it fails. That may be a conversion or a later consumer after an open field lookup produced `null`. Logs remain attributed to the node that discovered the problem.

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
}
```

The declaration records each field's contract:

- `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 prevent identifier syntax;
- `externalPayload` admits uncertainty exactly where it enters, without weakening the rest of the record.

Optionality, nullability, and defaults remain separate:

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

The `?` makes the property optional. `| null` allows `null` when it exists. `= null` records a schema default. A field may be required and nullable, or optional and non-null when supplied.

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 parentheses, `string | null[]` means either one string or an array of nulls.

The parser turns interfaces into JSON Schema metadata and carries referenced definitions. An `IncidentRecord` boundary becomes a Struct pin with that specific schema attached.

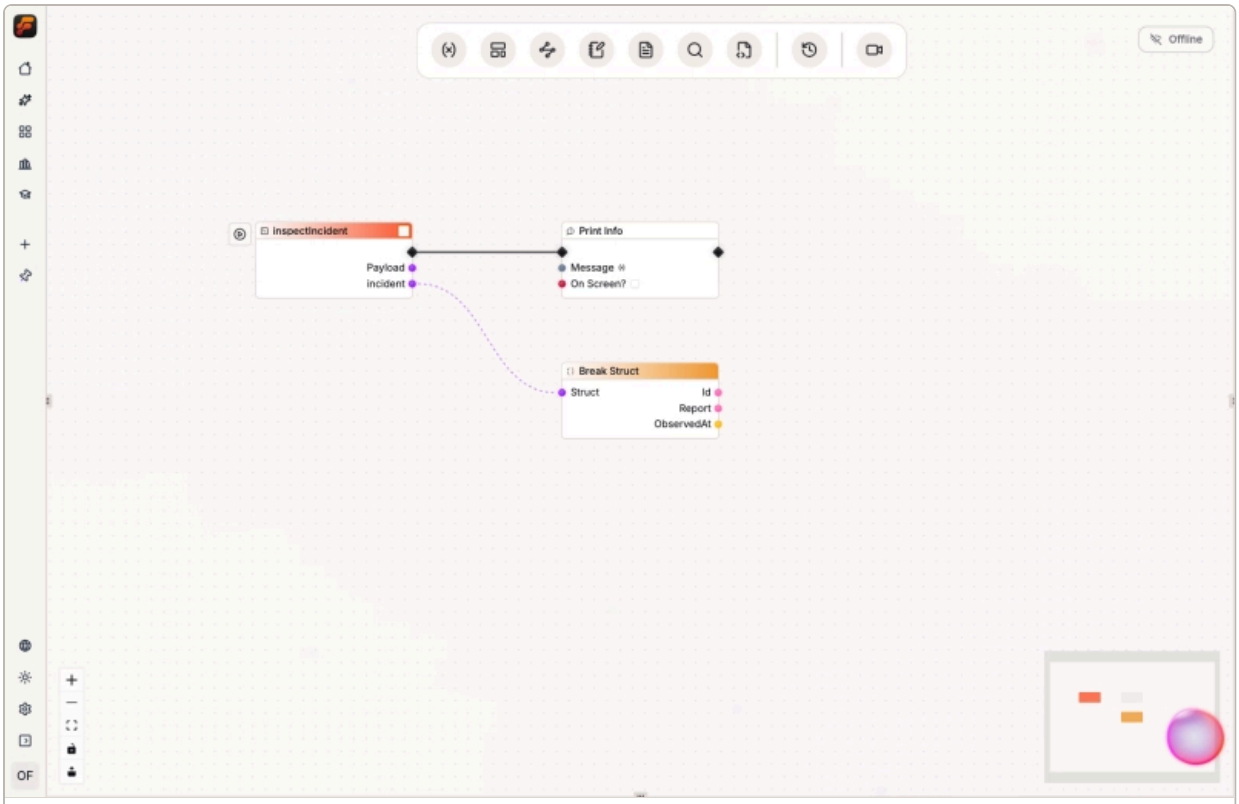
`IncidentRecord` describes fields on a wire. It creates no constructor, prototype, or arbitrary methods. Method syntax still resolves to a catalog node or FlowScript function with a compatible receiver.

7.5 Struct fields and schema propagation

Typing a Struct changes what the canvas can show.

An open `Struct` reveals no keys. Generic Get Field and Set Field nodes therefore need a string such as `"report"`, which may remain plausible after the remote field is renamed.

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 producer schema and creates the most specific output pin it can derive for each property. **Make Struct (Schema)** exposes field inputs from the consumer schema and produces the Struct. Field selection stays visible instead of relying on unchecked string keys.

FlowScript can render that Board logic compactly:

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

Identifier-like fields use dot syntax. Other property names use bracketed strings. Numeric or dynamic brackets remain collection indexes, so `incidents[0]` and `incident["external-ticket-id"]` have different meanings.

Depending on the live Board, the renderer may describe a Break Struct output or a catalog field-access node. Apply must resolve the expression to a compatible Board operation. Chapter 9 follows that lowering.

Schema propagation keeps field information after the first wire. Get Element and For Each carry a typed array's item schema, allowing downstream Break Struct nodes to expose fields. Nested Structs receive sub-schemas where derivation is possible. A generic passthrough preserves any concrete schema.

The current Make/Break implementation preserves a wired field pin when a new schema removes that field. The stale pin and connection remain, new fields appear, and the node reports the undeclared connection. The author can then insert a conversion or choose the renamed field without rebuilding a vanished wire.

Interface changes are handled according to 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 repair should appear where the change first becomes knowable.

7.6 Edges and preserved metadata

Type checks use known facts. The current behavior is:

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

WHAT FLOW-LIKE KNOWS	CURRENT BEHAVIOR
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` marks a missing base-type fact while preserving collection rules. An open Struct says its fields are unfixed; it does not match an unrelated concrete schema. Make and Break boundary pins are special because they adopt a connected Struct schema.

Schema compatibility does not provide full structural subtyping. Two enforced concrete schemas generally require canonical equality, except at schema-adopting boundaries. A three-field record cannot be assumed compatible with a two-field input. The node declares whether its schema is descriptive, enforced, or open.

The Board stores JSON Schema text or a reference to it. FlowScript projects the readable portion: object properties, requiredness, supported defaults, named definitions, arrays, string-keyed maps, supported unions and literals, `null`, `any`, and date or date-time formats.

Validation bounds, nested descriptions, pattern properties, and compositions such as `oneOf` or `allOf` may exceed that projection. The legacy `@schema("…")` decorator preserves an object schema that no named interface can represent, though it is hard to read.

When a Board contains a richer live schema, the renderer can show its interface projection while the reconciler retains the exact schema through an unchanged Board-to-source-to-Board round trip. A new standalone text file may not recreate hidden constraints omitted from that projection.

Top-level and function or event references currently support Normal, array, `Map<string, T>`, and `Set<T>` shapes around one base type. Interface fields also allow nested arrays, maps, sets, and unions. A Set projects to a JSON Schema array with `uniqueItems: true`.

Make/Break inference is narrower: an enum-only literal field may become Generic, and a map-shaped property may become an object-shaped Struct. The root interface remains intact, though derived field pins can be less precise. These are release observations.

An unresolved top-level, function, or event type name can currently fall back to a schema-less Struct, so a misspelled interface is not guaranteed to fail. Run publication examples through catalog-aware reconciliation; parser acceptance does not prove 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, report the smallest source, release, expected metadata, and actual result. Keep the limitation tied to that release.

These contracts guide authoring and repair on both the canvas and in FlowScript. Chapter 8 uses them to search and call the node catalog.

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.

FlowScript keeps its syntax small and puts most capabilities in the node catalog. Text, HTTP, files, databases, documents, models, and UI arrive as typed operations. Built-in and packaged nodes share one call surface: boxes with pins on the canvas, function or method calls in FlowScript, and named operations at runtime.

Every call resolves to a catalog declaration that the Board and run evidence can identify. This chapter shows how to find a compatible node for a known value and task.

Release check: Qualified calls, all four `use` forms, receiver dispatch, name-based output destructuring, generated declarations, editor completion, and context-sensitive search are implemented in the current repository. Ranking by safety, trust, permission, performance, or cost remains product direction. Section 8.5 records a migration gap: the general schema synchronizer can remove stale pins or detach changed-type pins without retaining a node error.

8.1 Qualified calls

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

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

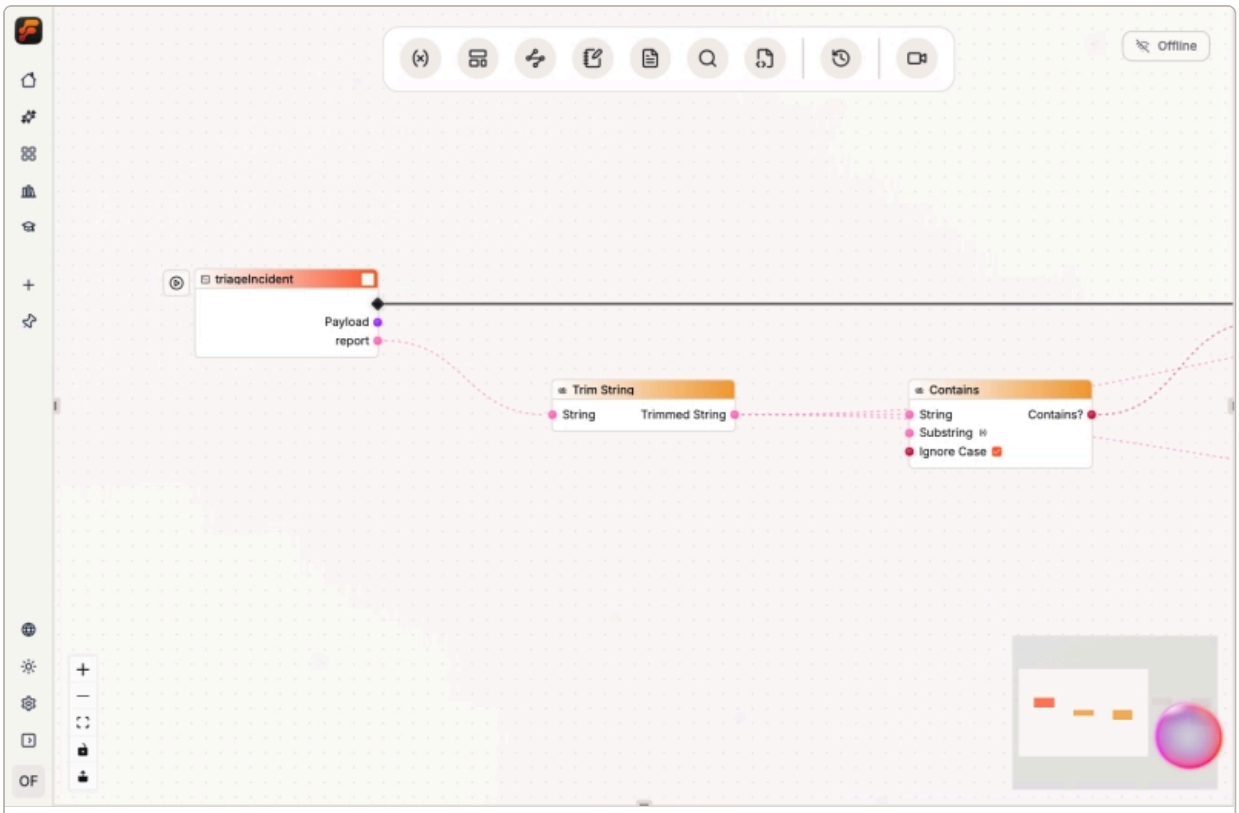
The path identifies a catalog namespace and member. The object maps values to input pins:

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

The first `string` is the namespace, `trim` aliases the Trim String node, and the object key names its input pin. `normalized` receives the sole data output. Reconciliation finds the live catalog declaration, plans that node, and wires `report` to it.

Catalog pins often use snake_case internally and render in camelCase, so `ignore_case` becomes `ignoreCase`. Completion inserts the projected name. Apply reports unknown, missing, or duplicated arguments and calls that match no declaration.

Use named arguments as the default model, even where positional forms are accepted. They expose the pin mapping:



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

For long signatures, the pin name also explains each value during review.

Generated declarations mark inputs with catalog defaults as optional. Required inputs need a literal, expression, or wire-producing binding. Only declared pins are valid arguments.

`::` joins namespace segments, as in `ai::ml::read`. A dot selects from or invokes on a value. The reconciler can correct some `string.trim(...)` mistakes, but source should use `::` for namespaces and `.` for receivers or outputs.

This Incident example uses several equivalent call forms:

```
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 trims a report such as `orders: production is on hold`, splits the system name from the detail, and logs whether the expected structure was found. Trim, Split Once, Branch, and both log operations remain visible nodes.

A qualified call resolves only against the catalog available to the App and release. It cannot install a package or grant permission. If the package is absent, Apply reports an unknown namespace, member, or declaration.

8.2 Imports and aliases

Top-level `use` declarations shorten repeated namespace paths within one FlowScript document. The resolved Board nodes stay the same.

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({ ... })`
`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. `text::trim` and `string::trim` resolve to the same node type; bare `warn(...)` resolves through its selected import.

Unknown namespaces, missing members, keyword aliases, and collisions produce diagnostics. An unused valid import produces a non-blocking correction. If two namespaces expose the same member, argument shape may select one; a remaining tie requires a qualified name.

Arrays and strings can both offer `contains`. Named inputs may resolve a bare call; `string::contains(...)` removes any doubt. User functions can also shadow bare catalog members or method aliases, which produces a diagnostic rather than changing the selected operation silently.

Imports are derived from resolved calls rather than stored as Board entities. Current lowering uses `use ns::*` for at least two static calls when no used member collides with a Function, another node's flat name, or an already opened member. Method calls do not count; a single call normally stays qualified.

After Apply, source may render with an equivalent import style. The Board stores node identity, not an author-written alias such as `text`, so canonical output may choose `string::trim` or `use string::*`.

For predictable source:

- begin with qualified calls when learning or resolving ambiguity;
- accept autocomplete's import edit when a repeated namespace becomes noisy;
- 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 allows the value left of a dot to supply the `string` input. These calls resolve to the same node:

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

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

Method form already binds the receiver pin, so supplying `string: normalized` again is an error. One remaining required input may be positional, as in `normalized.splitOnce(":")`. Use a named object when several inputs or options remain.

Method syntax adds nothing to a JavaScript prototype. The call still becomes a Split Once node with a String input wire, so duration, failure, permissions, and outputs remain attributable to it.

Receiver contracts power completion. After a `string`, the editor offers string and universal operations. Arrays get array operations; a titled Struct can get schema-specific and general Struct methods. Available packages participate through the active catalog.

An unknown receiver broadens completion. On `any`, `.contains(...)` might target a string, array, or package node. FlowScript uses argument shape and opened namespaces, then requires a qualified call if candidates remain tied.

Receiver metadata decides whether a node supports method form. A hash operation may declare a text or byte receiver even though its static name lives under `hash`; an author may also opt out. Check the generated declaration.

FlowScript Functions may use their first parameter as a receiver too. The call still targets a visible Function layer. In every method call, the dot binds one declared input pin.

8.4 Single and multiple outputs

FlowScript handles zero, one, or several data outputs by pin name rather than pin position.

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. A logger has no data output and renders as a statement:

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

For several outputs, select pins by name:

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

Before each colon is the output pin; after it is the local binding. `before` becomes `system`, `after` becomes `detail`, and `found` keeps its name. If a release renames a pin, reconciliation reports the missing output instead of binding by position.

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 Split Once output pins. Resolution checks declared node outputs before treating the expression as a Struct field read. Chapter 9 follows that distinction.

With several outputs, FlowScript looks for a conventional default named `result`, `value`, `output`, `out`, or `batch`. For a method-form transformation, it can instead select the output that matches the receiver stem, such as `map_in` to `map_out` or `array_in` to `array_out`. Other pins remain selectable. Split Once has no default among `before`, `after`, and `found`, so consumers must name an output.

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; pin names are the contract. Object destructuring survives reordering and maps visibly to data wires.

When one output has several consumers, every use resolves to the same output pin and the Board records the fan-out. The local name introduces no hidden state.

8.5 Generated `.flow.d` declarations

Flow-Like generates `.flow.d` files from node metadata so editor assistance matches the catalog used for reconciliation and execution.

An abbreviated declaration for Contains looks like this:

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

The declaration records:

- 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.

Node, pin, and output descriptions become hover documentation. Struct schemas live in a sidecar so tools can resolve fields without expanding JSON Schema in every signature. `names.json` maps node types, qualified names, aliases, receivers, receiver classes, and categories.

Files are grouped by top-level domain and source package. Studio's FlowScript editor uses the active catalog for completion, hover, signatures, and diagnostics. The VS Code extension reads the declarations, and FlowPilot queries the same index before writing source. All use the same pin names.

A declaration snapshot must match the App's catalog. Packages can add names or collisions, and upgrades can change pins. The parser does not bind calls from `.flow.d`; reconciliation uses the live catalog. Book examples therefore need catalog-aware reconciliation as well as parsing.

Nodes carry a schema version for migration. When the catalog is newer, synchronization matches pins by name, preserves IDs and wires for compatible types and shapes, adds pins, refreshes catalog metadata, and lets dynamic nodes rebuild derived pins. Widening a data pin to Generic preserves a compatible wire.

The intended rule is to migrate proven-safe changes and leave unsafe ones visible for repair. The node remains on the Board today, but the general synchronizer applies that rule unevenly. On an ordinary node it removes a pin absent from the new catalog. A type or collection-shape change clears connections and resets the default, then clears the node error. Some schema-driven nodes already retain wired stale pins and report an error, but this is not yet universal.

If a package or node type disappears, Studio keeps the node on the Board and marks it with an unavailable-package warning. This preserves its location and identity, but does not cover a pin-level breaking change in an installed package.

Until migration applies the intended rule everywhere, inspect package updates and affected Boards before treating an upgrade as automatic. Unsafe changes should stay visible for repair and must never alter a program silently.

8.6 A large library without a memory test

Catalog search should work even when the function name is unknown.

The checked-in declarations contain 1,668 node entries at revision `839395640`; packages and releases change that count. Type and intent metadata make the library searchable in context.

In Studio, opening Actions on empty canvas space supports browsing and text search across node names, friendly names, categories, descriptions, and pin names. Search supports prefixes and limited fuzzy matching. Without a query, results are primarily alphabetical.

Dragging a wire into empty space enables **Context Sensitive** mode by default. It keeps nodes with an opposite-direction pin compatible with the carried value, then search narrows the set. Choosing a result connects the

matching pin.

The filter considers direction, data type, collection shape, Generic specialization, schema enforcement, and schema-adopting Struct boundaries. Board validation still checks the connection. Turn Context Sensitive off to explore the full catalog.

FlowScript uses the same context at the cursor. After `normalized.` it offers String receiver nodes; after `string::` it lists namespace members. A bare call can offer an unopened member and insert the needed `use` line. Hover shows signatures, argument completion shows remaining pins, and constrained pins can offer allowed values. FlowPilot queries the same declarations by intent and returns exact live call forms.

Current visual search ranks textual relevance across names, categories, pins, and descriptions. It does not rank compatible nodes by package trust, permission breadth, quality, performance, or cost.

The planned direction is policy-aware ranking after compatibility. When several nodes fit, Flow-Like could prefer one that matches the organization's trust, permissions, performance, and cost policy, while explaining the ranking and keeping alternatives visible. Weighting, overrides, and provenance remain governance work.

Discovery follows the same loop on the canvas and in FlowScript:

```
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 node expresses the intent, compose lower-level nodes or add a reviewed package with a typed, capability-scoped contract. Later chapters show how WASM extensions enter an App and join the same catalog.

The next chapter shows how operators, templates, ternaries, and field access lower 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 can describe several nodes in a few familiar lines:

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

This describes a graph containing Integer Greater Than or Equal, Boolean And, Select, String Format, and Set Field nodes. Apply resolves each expression against the catalog and creates or updates the corresponding nodes, pins, values, and wires. No JavaScript engine evaluates it.

FlowScript may shorten a Flow only when it preserves the exact node contract. The renderer uses a short form only when it can reproduce that contract. Otherwise, the explicit call keeps every meaningful setting visible.

Release check: The current repository implements arithmetic, comparison and Boolean lowering, unary `!` and `-`, four compound assignments, value-selecting ternaries, template literals, and Struct field writes. It rejects mixed known operand types, although the diagnostic does not yet offer the intent-specific repairs proposed in Section 9.1. Operator and template rendering preserve configured extra pins in the cases below. Known asymmetries remain: Float equality and inequality may render as operators that text cannot reconcile; Integer division's declared result conflicts with its live node output; and a Struct Get `found` output may be mistaken for field-value sugar. These are implementation gaps.

9.1 Arithmetic, comparison, and Boolean expressions

An operator lookup uses its symbol and operand type. Given:

```
let urgent = attempts ⩾ 3
```

Because both operands are Integers, FlowScript selects Integer Greater Than or Equal and connects its Boolean output to the next consumer. 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 parser recognizes more tokens than this table lists. Apply still needs an unambiguous contract for the symbol and operand types in the active catalog. This is where syntax meets the catalog installed for the current Flow.

The parser recognizes `|`, though 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 because the 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.

Float equality usually needs a tolerance. The Float Equal node exposes that input, so `a == b` cannot carry the full decision. Choose it in an explicit call:

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

This keeps an operational assumption visible in both source and node. No global tolerance is silently inherited.

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
```

The meaning stays the same. The parentheses expose the expression tree and its operator nodes, without asking reviewers to recall precedence.

Types decide; FlowScript does not guess intent

The `+` symbol may identify Integer Add, Float Add, or String Concat. FlowScript chooses only when the operands determine one meaning. Matching String, Integer, and Float pairs use their respective families. An Integer literal may adopt the Float family when its other operand is a known Float, as in `ratio * 2`. Separately typed Integer and Float values require an explicit conversion.

Consider the deceptively small expression:

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

Two results are plausible:

- numeric addition after parsing the left side: `6`; or

- text concatenation after formatting the right side: `"51"`.

An implicit conversion would choose a business decision for the author. Reconciliation reports incompatible `String` and `Integer` operands and creates no String Concat node. Apply is atomic, so the failed expression leaves no partial 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 intended editor experience rejects the ambiguous Apply, offers both repairs with node previews, and lets the author apply one. Canonical source then shows that choice. Conversion policy, including failure, truncation, fallback, or representation, belongs in the Flow.

The same applies to `any + 1`, which has no reliable operator family. Narrow it through a typed interface, use a parser such as `string::toInt`, or place `types::tryTransform` and consume its `success` result. Try Transform is a catalog node whose Generic output adapts to its typed consumer. An untyped or unconnected output produces `null` and `success = false`; failed conversion is an ordinary result. Ignoring `success` moves 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 Boolean Not, which canonical source generally shows as a catalog call. Numeric negation canonicalizes to subtraction from zero:

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

The reconciler chooses Integer Subtract or Float Subtract from `balance`'s type. A negative literal such as `-3` stays a literal; `-balance` becomes an operation visible on the Board.

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
```

The expanded form reads the current producer, creates the next value, and rebinds the name for later statements. The graph contains no opaque CPU-style `+=` instruction.

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 remains Generic until the Flow proves more, so the author may need an explicit read and conversion.

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.

Select evaluates its condition and reads only the chosen data input. It exposes no alternative Execution arms. Use a ternary for value 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 covers those paths. An impure expression still needs a place in the surrounding execution chain.

The condition must be Boolean, and both alternatives must fit the selected output contract. When schemas or container shapes differ, define an explicit common boundary instead of erasing the difference through Generic.

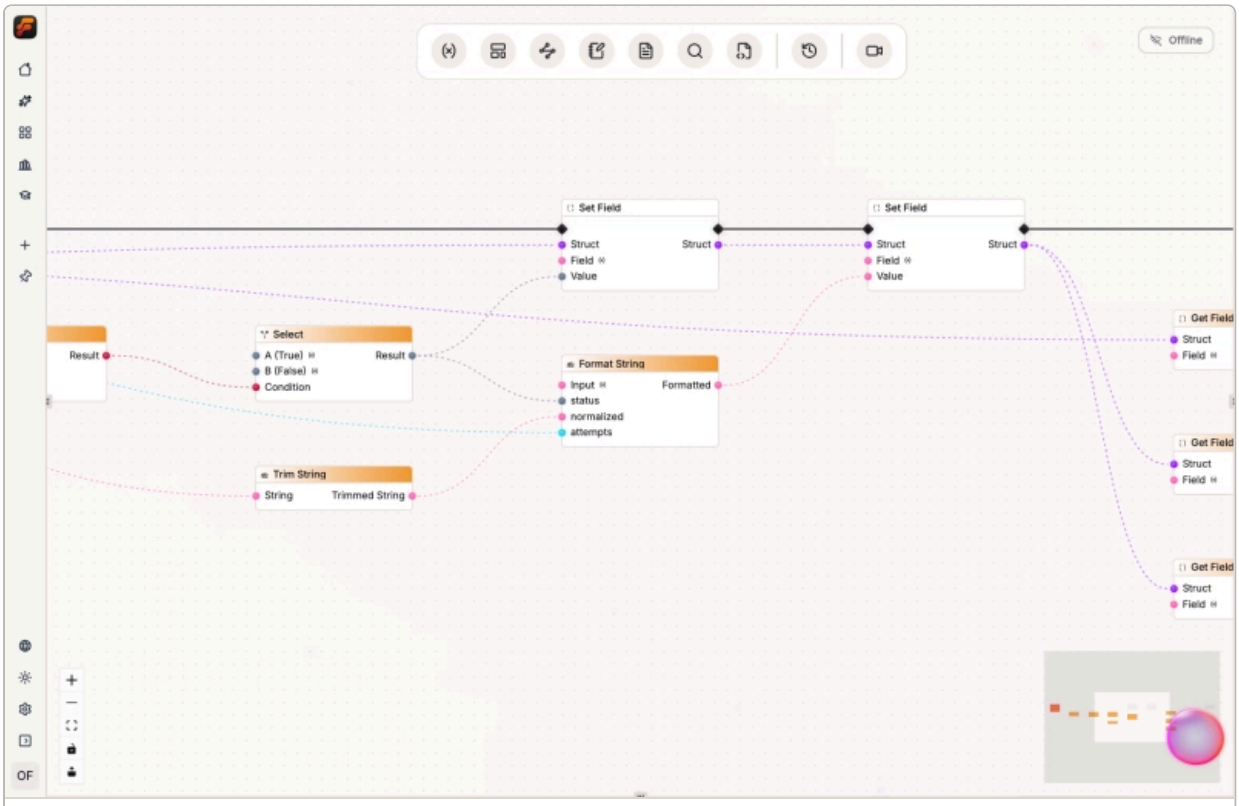
The renderer recognizes `utils_types_select` specifically. Its current `condition`, `a`, and `b` inputs fit the ternary exactly. Unlike binary operator lowering, this path does not yet guard against future configured inputs; the lossless-sugar test must grow if Select gains one.

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 node pins, so every placeholder has an inspectable producer.

Simple references keep their names. Member or output access normally uses its final segment. Complex expressions receive stable names such as `arg1`; collisions gain a suffix. Repeating the same bare reference reuses one placeholder pin.

Dynamic pins are part of the node. The Board must know which wire supplies `{status}`, and rebuilt text must produce the same pin set.

Literal text shaped like a placeholder, such as `{status}`, is ambiguous because String Format would interpret it as a pin. FlowScript rejects that template. Use an explicit format call for a placeholder; change the text when the braces should be literal, since the node cannot preserve an escape.

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,
})
```

The explicit call prevents a dynamic pin from being renamed or a configured value from being dropped.

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 `split`; the second reads a Struct field. Reconciliation checks declared node outputs first, then uses a Struct field-access node when the base is Struct-shaped. A named interface retains the field's declared type, while an open Struct produces a Generic boundary.

A current round-trip bug affects Generic Get Field, which exposes `value` and `found`. Board-to-text lowering recognizes member access before checking the wired output, so `found` may render as `incident.status`, which normally means the value. Until lowering checks the pin, use an explicit call and named output when the presence flag matters.

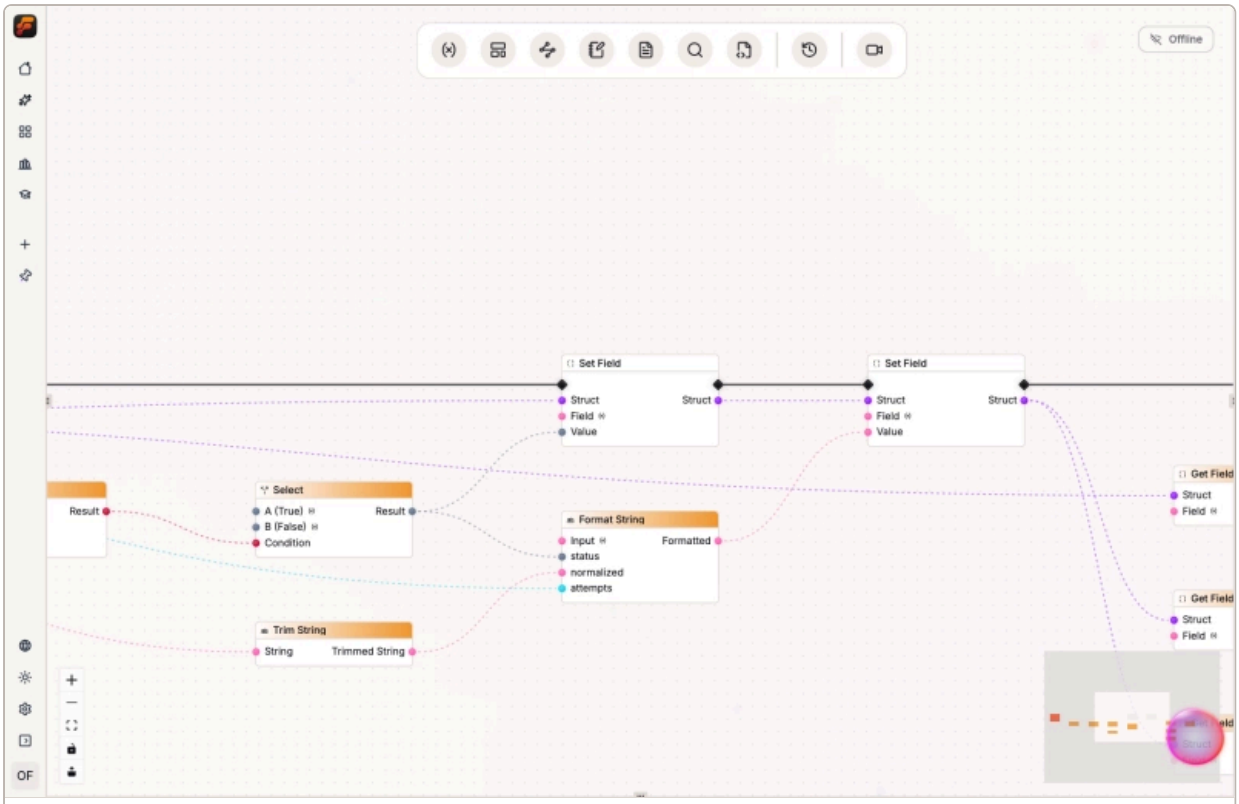
```
const { value: status, found } = incident.get({ field: "status" })
```

A field write is equally concrete:

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

It lowers to Set Field with `struct_in`, the literal path `"status"`, and the new value as inputs, plus a `struct_out` output. FlowScript 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 how source-level Struct mutation works. A consumer wired before the assignment keeps `incident@0`. References after successive writes receive `incident@1` and `incident@2`. The source name selects the producer used by later expressions. Each operation's pins carry the runtime value for that version.

An unused updated Struct is simply an unused copy. Existing wires still point to their earlier producer.

Nested literal paths follow the same model:

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

A computed field requires an explicit Set Field call. The renderer uses dot assignment only for an accumulator chain whose output becomes the next value of the same binding. Seed operations, cross-source updates, and wired dynamic fields remain explicit.

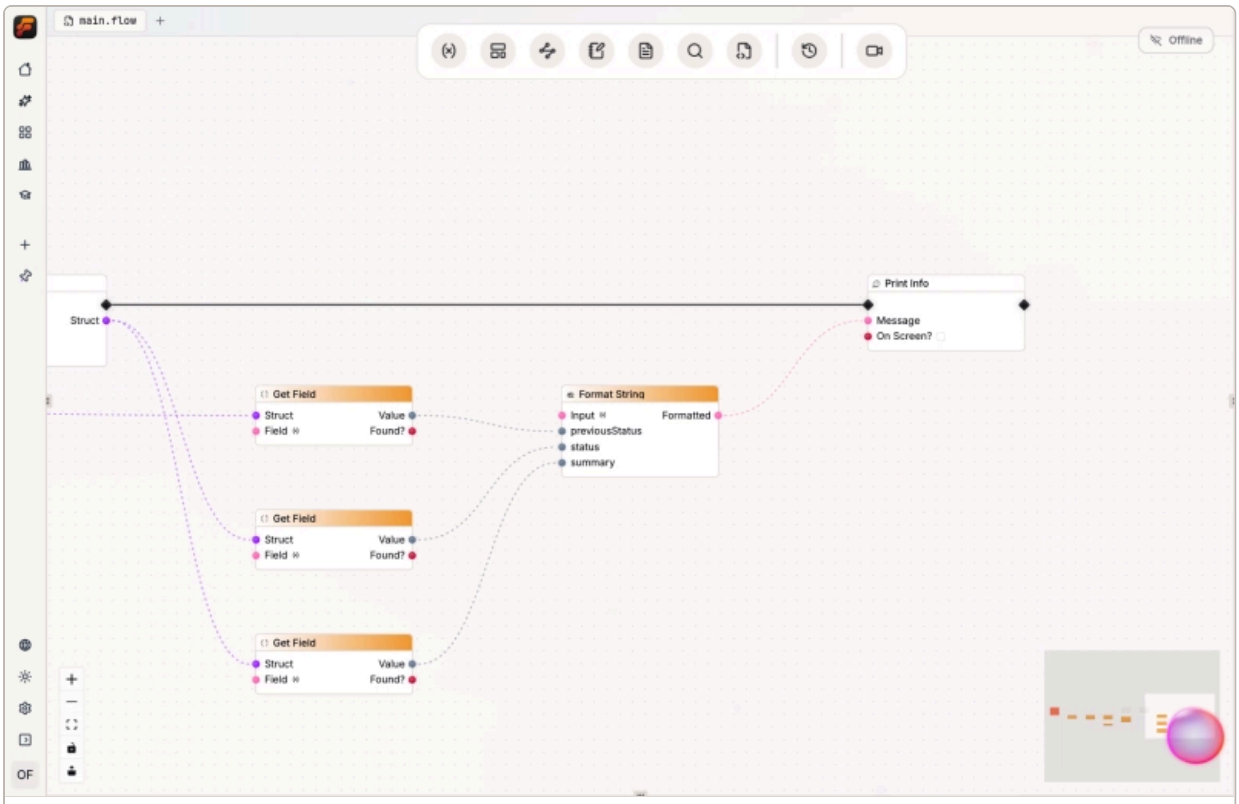
9.6 Sugar must round-trip honestly

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

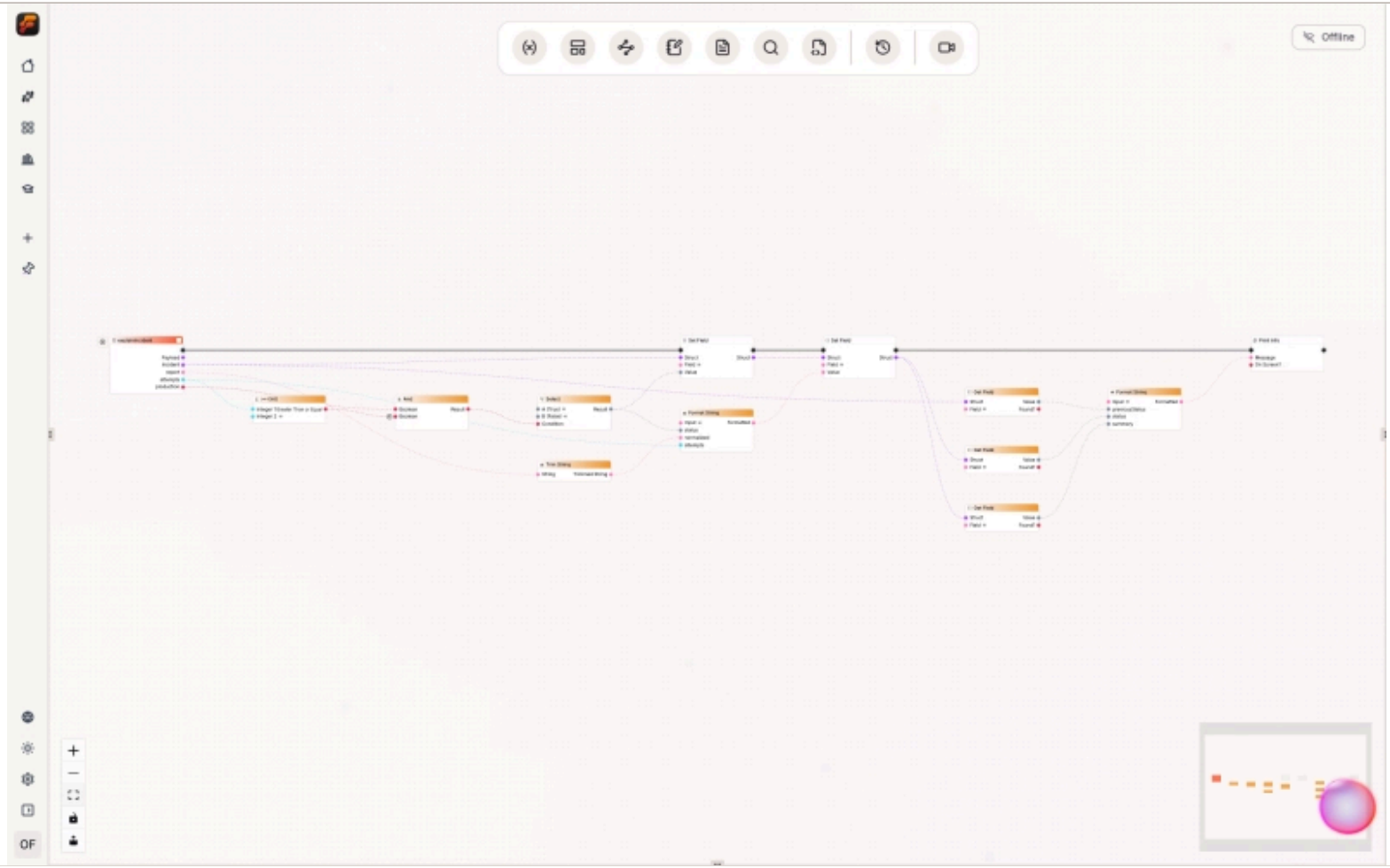
```
use log::{ info }

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 })
}
```

The Flow trims a report, classifies repeated production trouble, remembers the old status, produces successive Incident values, and logs the transition. The graph shows the same logic:



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.

Board layout may change. Every meaningful source operation still has a node or pin value, and every meaningful configured input should survive when the Board becomes text. The release gaps above mark current exceptions.

Board-to-text lowering checks more than 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 make source longer when that is required to preserve the graph.

In the chapter example, the first Set Field may read back as an explicit seed alias, possibly named `record`, while only the next same-accumulator update returns to `record.summary = …`. The fixture shows canonical source, not guaranteed byte-identical Board readback. The graph and configuration are the contract.

An invalid variant makes the boundary clear:

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

The parser recognizes the expression, but Apply cannot choose a node family. It reports the String/Integer conflict without inventing a conversion. Use formatting for a label, or parse the String and handle success for arithmetic.

Compact notation stays only when it carries the entire node contract. Otherwise, FlowScript shows the explicit node call.

The next chapter applies the same standard to `if`, named outcome arms, loops, `while`, and `return`: every execution path must have a 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.

FlowScript maps familiar control syntax to visible execution paths:

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

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

An `if` is a node with execution outputs. A collection loop owns an iterable, exposes value and index, and triggers its body path for each item. While owns a condition and iteration limit. The statement after either loop connects to **Done**. A `return` wires data to a function boundary or terminates one Event branch.

The canvas therefore carries runtime meaning. Node position has no effect on execution. A changed wire can change which work runs and what follows it.

Every path that can run, fail, repeat, or finish must remain visible in source and on the Board.

During the Chapter 1 incident, the visible Flow shows the decision, attempted call, expected Error route, unexpected failures, and recovery work at the responsible nodes.

Release check: `@parallel` currently sets a concurrency limit of 30, and plain `while` permits at most 15 iterations. FlowScript has no `break` or `continue`; function `return` does not yet terminate the whole function. Loop control nodes can log and absorb child-path errors, so work may continue and a run may end with Success despite Error evidence. This remains an implementation gap. The sections below mark the behavior of each loop.

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 Boolean condition feeds a branch node. Its blocks connect to True and False execution outputs, and only one activates. Position on the canvas cannot change the choice; wires and data do. The condition itself is data flow into the branch, so failures while producing it remain attributable to their source nodes.

Chapter 9's ternary makes a different kind of choice:

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

A ternary selects a value. `if` selects a path, which suits calls, writes, retries, tickets, and other work that should run on only one 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 nested form matches the topology: the second decision is reachable only through the first branch's False arm. Its indentation shows that ownership.

A branch is not an error handler

A False arm means the condition evaluated to false; it catches no error from the condition or True arm. Expected negative outcomes need a modeled execution output. Recoverable unexpected node failures use **Handle Errors**. The Board shows these contracts separately.

10.2 Named execution arms

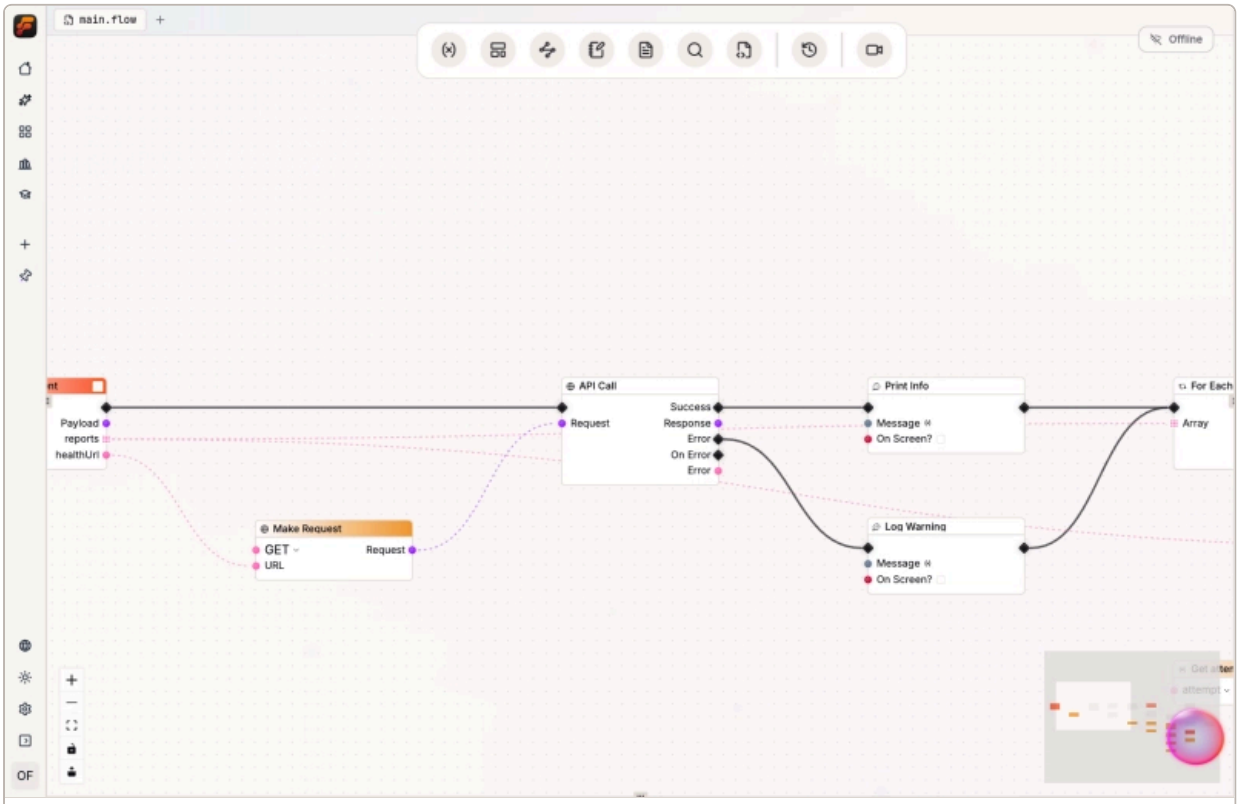
Some decisions need labels beyond True and False. The HTTP API Call node exposes Success and Error, which FlowScript preserves in a named arm block:

```
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 that can expose data such as `apiCall.response`. The block connects work to its execution pins. Nodes with more outcomes use one named block per connected arm.

For the current HTTP node, a completed 2xx response selects Success and a completed non-2xx response selects Error. Transport failure, an invalid request, or a response-read failure can raise an unexpected node error. **Handle Errors** adds an `On Error` path and Error string output for that category.

The Board keeps both routes visible:



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. If an On Error recovery path completes, execution can continue while the original node retains failed evidence.

This separation matters in operations: an HTTP 503 is an expected protocol outcome, while a malformed request value indicates that the node could not perform the call as modeled. They need different recovery policies and traces.

Arm names belong to the node contract. Canonical source camel-cases internal pin names, such as `exec_error` to `execError`; Apply also recognizes supported raw and friendly spellings. Use Board rendering, editor completion, or Apply's available-output diagnostic for the accepted name.

Some two-output nodes render as `if (nodeCall())` with comments such as `// exec_out_exists` and `// exec_out_missing` after the braces. These semantic labels preserve the graph output represented by each arm.

10.3 Sequential `for … of`

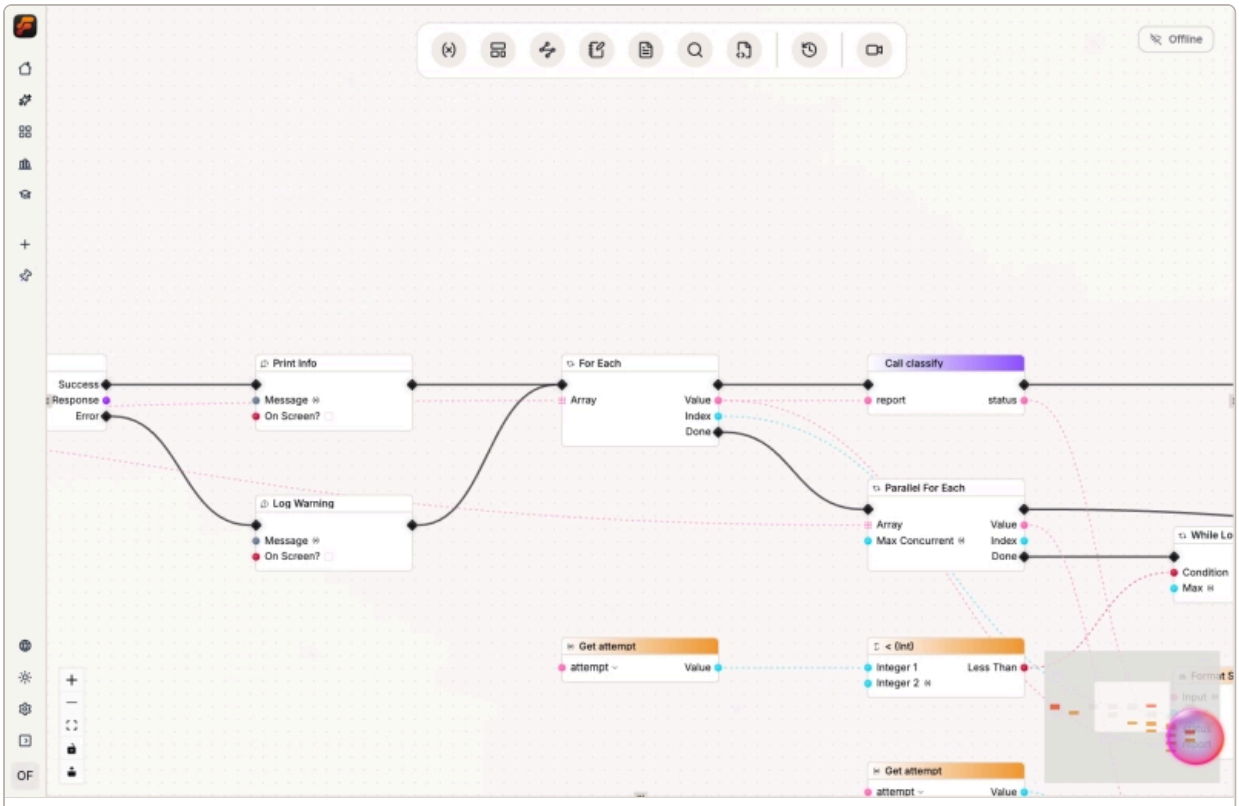
The ordinary collection loop is sequential:

```
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 and processes values in order. It publishes each value and index, then waits for the body chain to settle before starting the next item. **Done** activates after all items and starts the following statement.



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

Use ordinary `for` when iterations touch shared or external state: rate-limited APIs, ordered or deduplicated tickets, dependent database writes, bounded model spend, or any operation whose idempotency remains unproven.

Sequential here describes scheduling, not outcome policy. The loop still owns every iteration and decides how execution continues after a child path settles.

Sequential loops are not fail-fast today. When a body chain returns an unhandled error, that chain stops. The loop logs an Error under its own node and names the iteration, then continues. Logs from the failed chain retain child-node attribution. A modeled or handled failure can create a ticket before later items run, which suits batches where one bad record should not discard the rest.

The loop node returns success after logging a child error, so the run may report Success despite a failed child trace. The intended invariant marks the aggregate run Failed after remaining work settles. Until it is uniform, a Flow that must report “all records succeeded” should collect per-item outcomes and make the final decision visible.

10.4 `@parallel for`

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

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

Parallel For Each starts independent items up to its configured concurrency limit. `@parallel` is lossless only at the default of 30; a custom limit remains explicit:

```
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; with one body root, it matches active iterations. `0` means unlimited, and current code treats every non-positive value that way. Avoid unlimited concurrency for external services.

Done is a barrier, not an ordering guarantee

Parallel For Each activates Done after every child settles. It exposes no collected-results array, and effects may finish in any order. When order matters, carry the original index, collect results, and sort them. Gather provides an execution barrier without collecting data values. Node position, log arrival, and apparent finish order reveal nothing about result order.

A failing child does not cancel siblings. The node schedules remaining items, drains child work, logs each failure, activates Done, and returns success. The sequential loop's aggregate-status gap also applies.

Before opting in, verify that order is irrelevant, writes are idempotent, retries cannot duplicate effects, downstream rate and connection limits are known, concurrent model/network/memory cost is bounded, failures are

collected, and sibling behavior after failure is acceptable. Keep external calls, ticket creation, and database writes sequential until that contract justifies overlap.

Already-running siblings are especially important: the current node does not cancel them after a failure. Any side effects they start must remain safe even when another item has already failed.

10.5 Bounded `while`

Flow 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 its condition dependencies and evaluates the Boolean. The current node also has a `maxIter` guard. Plain `while (condition)` keeps the default of 15; a custom guard makes the node call explicit. The body starts only when the refreshed condition is true:

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

The maximum bounds body starts. It cannot prove progress, enforce a wall-clock deadline, add delay or backoff, or cancel a slow body operation. Model those policies in the Flow and called nodes.

If the condition remains true at the maximum, While silently activates Done. It has no Exhausted output and emits no warning for reaching the ceiling. When exhaustion means failure, maintain an attempt value and check it after the loop before choosing Success, Retry Later, or Escalate.

Body failures are logged; the loop reevaluates its condition and may finish successfully. Direct condition evaluation failures can propagate. Failure to retrigger a dependency is currently logged before the loop exits

through Done. Test these release-sensitive details before relying on them.

Cancellation is observed at node boundaries. Long-running nodes must cooperate with it themselves. Plain loop nodes observe cancellation less consistently while walking or scheduling iterations than batch-loop variants. Keep an iteration maximum and external-operation timeouts even when callers can cancel.

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 empty False arm reaches the body end and advances the loop. The Board shows the skipped path.

The catalog's **For Each (Break)** node samples a Boolean Break input before the loop and after each body root. True stops remaining roots and items, then activates Completed. This visual node is outside FlowScript's structured loop registry, so it does not round-trip as `break` or ordinary `for` sugar. Use it explicitly and verify rendered source against the target release.

This authoring gap needs a future text form that preserves the stop condition and Completed path.

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 expressions wire positionally to `status` and `normalized`. Literals can become typed values, and a bound call output can supply a return pin. Validation reports extra return values or output pins without sources instead of guessing at an arity mismatch.

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()
```

Current function reconciliation wires data to layer output pins and has no function-wide Return node. FlowPilot's intermediate representation rejects nested function returns and permits one final, unconditional top-level return. For early selection, branch to compute the result, rejoin, and return once at the boundary. This makes every declared output's source visible before the function completes.

An Event return has a different implementation:

```
eventsGeneric status(payload: Struct) {
    return "accepted"
}
```

This becomes a terminal Return Result node with no execution successor. It ends that branch and publishes one Event result. Events accept one return value; use a Struct for a compound response.

An Event return does not cancel sibling branches. Execution surfaces also aggregate competing results differently. Synchronous server and remote callers usually keep the first; subcontext merging overwrites with the last

merged child; UI run state shows the last received result; SSE forwards every result. Parallel timing makes implicit selection unreliable.

These differences are observable API behavior. A Flow with several result paths may answer differently depending on how it was invoked, even when its Board is unchanged.

Produce one logical result path per invocation. Never let racing `return` statements choose an answer by timing.

When branches produce different outcomes, join their data and select once before the result boundary. This also eases migration to a future 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"
}
```

On the Board, the Event opens a path and API Call splits it into named outcomes. Because both arms continue, the following statement fans in from both tails; HTTP 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 ends the Event branch and supplies the caller's value.

The canonical fixture also includes `@parallel for` and bounded `while`; the repository's FlowScript parser/renderer test checks it. It stays small enough to inspect in both views. Move larger logic into visible layers and functions.

FlowScript uses familiar control syntax only where it maps faithfully onto the Board.

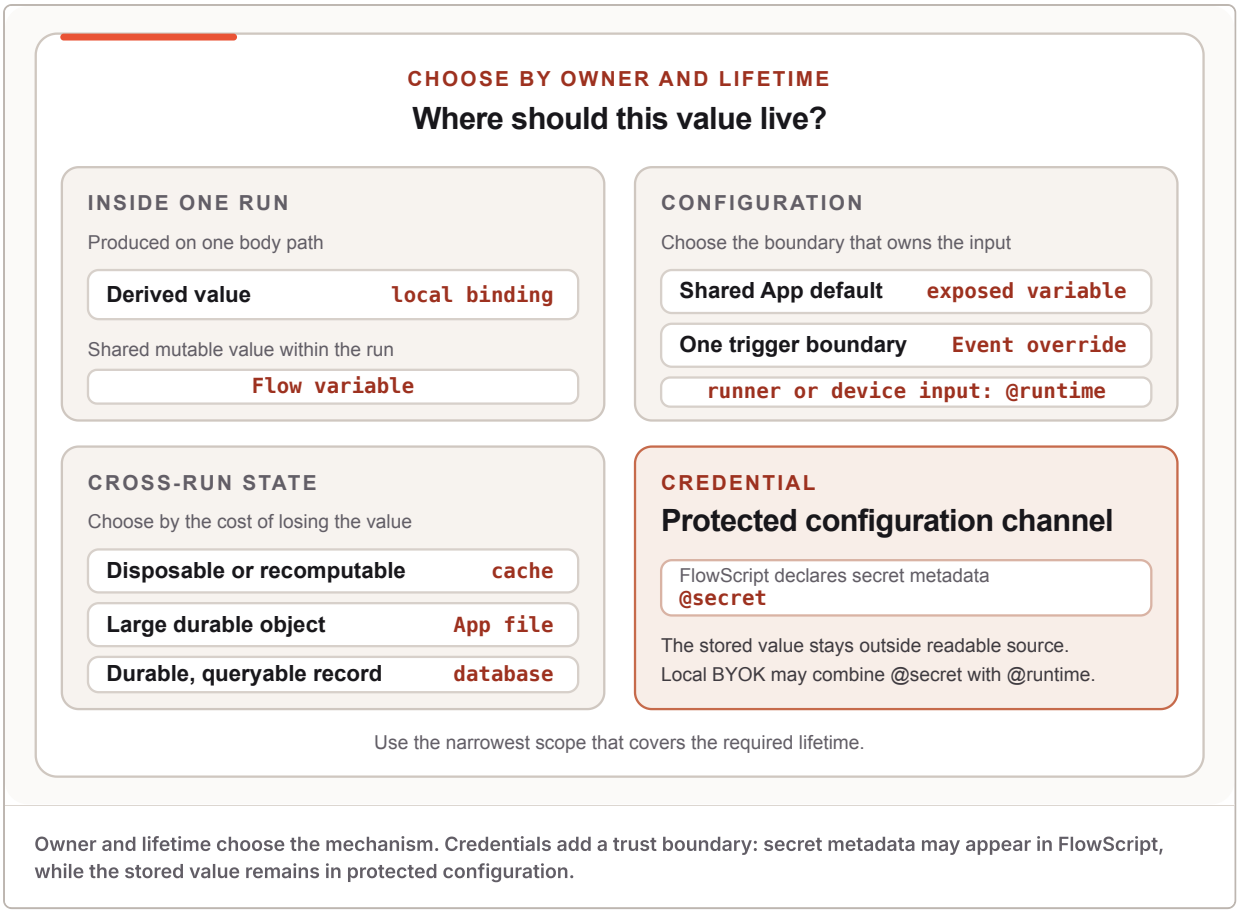
PART II · CHAPTER 11

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 and retry limit are configuration. An OpenAI key is a credential. The last successful synchronization may be durable state, while a ten-gigabyte dataset needs storage built for its size. Calling all of them “values” hides the important differences.

Before choosing syntax, ask **who owns this value, and how long must it remain true?** Keep state in the narrowest scope that covers its lifetime. Durable facts need durable storage, and credentials stay out of source.



Owner and lifetime choose the mechanism. Credentials add a trust boundary: secret metadata may appear in FlowScript, while the stored value remains in protected configuration.

Release check: Each run receives fresh mutable Flow variables. Top-level `const` means non-exposed and `let` means exposed; neither is immutable. `@runtime` and `@secret` can be combined. Web and Desktop store values in local IndexedDB by App and variable, so they are **device-profile local**, not explicitly user-keyed. Interactive paths prompt for missing values, while direct and unattended paths may still use a Board/Event default or `null`. `@readonly` disables editing surfaces but does not block Set Variable during execution. Per-user/device identity, universal preflight, and runtime `@readonly` enforcement remain incomplete.

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` names the String Trim output; `urgent` names the Boolean from String Contains. They are graph bindings, so consumers connect directly to those producer pins.

A mutable local can model an accumulator:

```
let attempts = 0
attempts = attempts + 1
```

When paths need shared mutable state, the reconciler can materialize a local variable and Get/Set nodes. It lasts for one run; the next Event starts from configured defaults.

Top-level declarations describe Board variables:

```
const urgentPhrase = "production is on hold"
let retryLimit = 3
```

Generated Get and Set operations access these variables throughout the Flow. Each run has its own map, so concurrent invocations do not communicate through `retryLimit`, and writes do not change the next run's declaration.

This isolation is useful for parallel invocations: one incident can change its in-memory retry limit without altering another incident already in flight.

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.

Anything that must survive an invocation, such as sync history, balances, inventory, or job state, belongs 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 editable, exposed `let` appears in App Configuration. Authorized people can change it without opening FlowScript or rearranging the Flow. Events may carry permitted overrides. App permissions decide who can make either change.

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 chooses what is configurable; App configurators set shared defaults. Knowing a non-exposed variable's ID must not grant a caller permission to rewrite it.

Changing an exposed default changes Board configuration for everyone. Use runtime configuration for a personal or device-specific language. Configure production and development at their Event or deployment boundaries instead of detecting the 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

Use a named interface when source should describe a Struct contract; `@schema` is legacy metadata. Decorators sit immediately above the declaration. Canonical order is description, category, legacy schema, secret, readonly, 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. UI edits are disabled, yet a Set Variable node can still change the value during a run. Until runtime enforcement exists, `@readonly` provides neither a security boundary nor an integrity guarantee.

Top-level `const` controls exposure; `@readonly` expresses mutation policy. A hidden value can be mutable, and an exposed reference value may need to be 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 contract while Web and Desktop store the configured JSON value outside it in local IndexedDB. A saved value overrides the Board default for that run. Remote payloads may include non-secret runtime values.

The intended scope is per user and device. Today's key contains App ID and variable ID inside the browser or Desktop profile. It isolates profiles, but cannot distinguish two Flow-Like accounts in one profile. A strict user-and-device promise requires a different key.

Treat the current store as local application state tied to that profile, not as an account-level configuration service.

Interactive execution currently follows a useful preflight:

```
Run requested
    │
    ├─ saved records present ─▶ execute
    │
    └─ values missing ─▶ configuration dialog ─▶ save ─▶ execute
```

A required runtime value should be valid before any node runs, whether execution is interactive, scheduled, API-driven, internal, or agent-initiated. The interactive service currently checks only that a record exists. Core runtime does not enforce universal presence; without an override, it may resolve Event configuration, the Board default, or `null`. Universal preflight remains future work.

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 has no value. FlowScript rejects a non-empty secret initializer, and rendering omits stored secret values. Opening, copying, sending, or editing source therefore reveals none.

For a local interactive run, Desktop can prompt once and save the value locally. The masked field can be revealed deliberately. Device-local storage alone does not promise encryption at rest, so the operating-system account and application data still need protection.

Standard clients remove local secrets from remote payloads. Remote BYOK therefore needs trusted server-side Event configuration. Event reads return blank secret values, and unrelated edits preserve stored values. Deployment determines at-rest protection; masking alone says nothing about it.

Filtering also prevents a browser or Desktop client from casually forwarding a device credential to a hosted executor.

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,
})
```

The Flow declares the credential contract, a trusted configuration surface receives the value, and the provider node consumes it without source embedding. Local storage does not make that key available to every remote execution mode.

For hosted OpenAI BYOK, use a private provider/model profile. The server encrypts its credential, keeps it apart from public model metadata and ordinary profile responses, and hydrates it only in the trusted execution path. This also avoids copying a key into every Flow. Offline Desktop must download the credential to local settings, so device-profile protection still matters.

Secret metadata reduces exposure, but no universal redaction pass can recognize credentials copied into arbitrary messages. Never log, ticket, interpolate, or return a key.

Changing a hidden secret safely

The editor cannot validate a hidden value. FlowScript will not change a secret's type, container shape, or schema while preserving an unknown default. To declassify or replace it, first use the guarded clear/declassify path, then make the ordinary type or value change explicitly.

Source and AI tools may change the credential contract without gaining authority to read or write the credential.

11.6 Choose the right lifetime

Choose by the cost of losing a value. Use this 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

The key-value cache supports App and user scopes, namespaces, expiration, deletion, and values of roughly one mebibyte or less. It is separate from the evictable file cache directory. Cache `lastSuccessfulSync` only when loss causes a safe repeat sync. If loss could skip work, repeat an irreversible effect, impair recovery, or break an audit obligation, use a durable database record with the required retention policy.

For reference data, use an App file when the dataset is large, versioned, and mostly read-only. Cache small derivatives that can be recreated. Use a database for records that are updated, filtered, joined, governed, or changed concurrently.

One application may combine a source file, normalized database records, and a hot lookup cache. They still share the same App, permissions, runtime, and evidence model.

Flow-Like's native data path can retain storage versions, but optimization may prune them. Set cleanup and retention policy deliberately. Regulation or recovery that requires an immutable journal needs an explicit audit model.

A local chat session belongs to one conversation. Current "global" chat state spans chats for the same App/Event on one local client. Use it for conversation continuity, never organization-wide or cross-device truth.

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 omits `lastSuccessfulSync` and reference data from global variables. Their longer lifetimes require appropriate 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.

Copying a useful node network creates a maintenance question: may the copies diverge, or must they stay identical? If they must stay identical, make them a function.

An explicit cache policy can also reuse a function's previous result. An ordinary visual group has no such call contract.

These constructs may look similar on a canvas, but serve different jobs:

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
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 become Function layers with typed boundary pins. Their purity follows the structure of their contents; each node designer declares node purity through the presence or absence of Execution pins. Any user function with a parameter can be called as a method on its first argument. `@cache` is explicit and may be attached to an impure function; a hit skips the whole body. Cache keys omit the function body, package versions, global and runtime variables, model choice, and external state unless those dependencies become inputs. A Function layer cannot yet be registered directly as an agent tool. It needs a referenceable Event handler; the planned automatic shim is not 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 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 part of the visible graph contract. Renaming, reordering, or retyping a parameter changes a pin. Changing a return changes what callers can consume. All callers share the same implementation.

Use a function when:

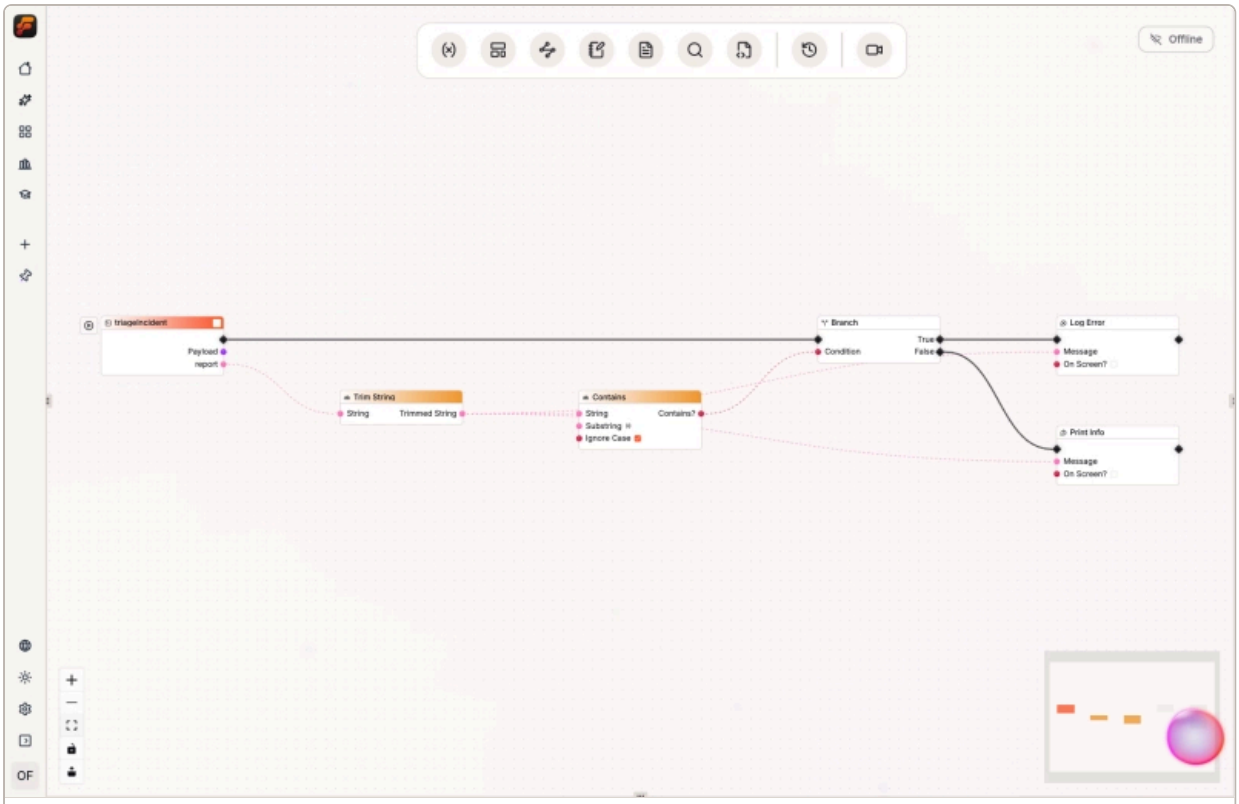
- several callers need logic that must remain identical;
- the operation needs a named, typed contract whose internals may change;
- the result should be tested as a unit; or
- the call needs an explicit cache policy.

Do not extract every cluster. A one-off group of nodes may read better inline, especially when its wires explain more than a generic helper name would. Extract it when the boundary clarifies the logic.

Functions are reusable within their Flow. Public APIs, schedules, and agent tools need a triggerable Event boundary; cross-Flow reuse needs a packaged node.

12.2 Execution moves forward; pure data is pulled backward

Flow-Like evaluates in two directions:



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 walks backward through any pure data dependencies, evaluates them in dependency order, then runs the consumer.

This gives an apparently small rule a large consequence:

> *An unused pure value performs no work.*

If `systemId.trim()` reaches no return pin or executing consumer, it does not run. An impure call on the execution chain does run even when nobody uses its data output.

Who decides purity?

The node designer does. The runtime uses this structural test:

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. The engine does not inspect Rust or WASM code to prove determinism or the absence of side effects. Hiding a network request or mutation behind a pure data pin breaks the execution model.

For node authors:

- make inexpensive deterministic transformations pure;
- make state-changing operations impure;
- usually make expensive work impure so its cost and schedule stay visible.

FlowScript derives a Function layer's execution shape from its body. An impure call, Board variable write, branch, or loop gives the Function execution boundary pins. A function containing only pure data dependencies and declared returns can remain pull evaluated.

"Deterministic" and "structurally pure" differ. The cached `resolveSystem` example contains an `if`. Its result is deterministic and side-effect-free, yet the planner represents its branch with execution flow, making the Function structurally impure.

The Unreal Engine comparison

Blueprint authors will recognize the model. 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 uses the same split between explicit execution and demand-driven data. If operators need to see when and where an expensive calculation runs, mark it impure even when it does not mutate state.

Purity answers **when is this data needed?** Caching answers **may an older result replace this call?**

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 method form, `systemId` fills the first parameter and the remaining arguments follow:

```
const { team, runbook } = normalized.resolveSystem(directoryRevision)
```

This works for every user function with a first parameter. Its type gives the editor an unambiguous receiver contract for suggestions after `receiver.`.

Method syntax is call sugar over the same pins. It does not imply object ownership, receiver mutation, or a different runtime. Board-to-source projection may normalize it back to a flat call, so the spelling is not yet preserved across a round trip.

For catalog nodes, the designer selects a receiver pin in catalog metadata, with a limited first-data-input fallback for compatible namespaces. Having an input alone does not make a catalog node a method.

12.4 Functions are reusable; Events are triggerable

A function has a typed call boundary and no independent runtime entry. An agent needs an Event to trigger it.

Any suitable Event can be registered explicitly as a tool. When the reusable logic lives in a function, a thin Event shim supplies the 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 reference metadata rather than an array passed through a data pin. It records the concrete handler the agent may invoke. The nested handler remains an independent Event entry. Its parameters become inferred tool input properties; its `return` becomes the result.

The model-facing schema describes those properties, but does not mark them required or advertise a declared output schema. The handler must validate its inputs.

Today, the author must write this handler. Registering `resolveSystem` directly fails because a Function layer has no trigger node. A future convenience may generate the shim while leaving the Event and registration visible.

Tool registration accepts referenceable Event entries, including the supported Simple, Generic, Chat, and Widget Action entries. It does not accept every start node. Registering an Event for an agent does not expose a public API; Chapter 13 covers that boundary.

Tool schemas are inferred; policy remains authored. Function references carry no confirmation, allowed-caller, cost-limit, or permission object. Put required confirmation or domain authorization inside or next to the handler. Package capabilities and runtime permissions still apply.

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 outputs, activates the continuation, and skips every node in the function. Side effects in the body do not happen.

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 trusts the author. Apply permits caching a structurally impure function, though the UI can warn. This may suit an expensive read or deterministic branch. Do not cache behavior that must occur on every call, such as a write, notification, or audit entry.

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; array order is preserved. The stable Function layer ID prevents functions in one namespace from sharing 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 changes the legitimate result, expose its identity or revision as an input, or invalidate or version the namespace. That is why the example accepts `directoryRevision`.

Bare `@cache` currently means namespace `global`, a five-minute lifetime, and App scope. The visual Function editor has different defaults, including no expiry. Until the editors converge, the book uses a settings object with an explicit namespace and TTL. App scope is omitted because it is the FlowScript default. Reserve `ttlSeconds: 0` for entries with an invalidation and cleanup plan.

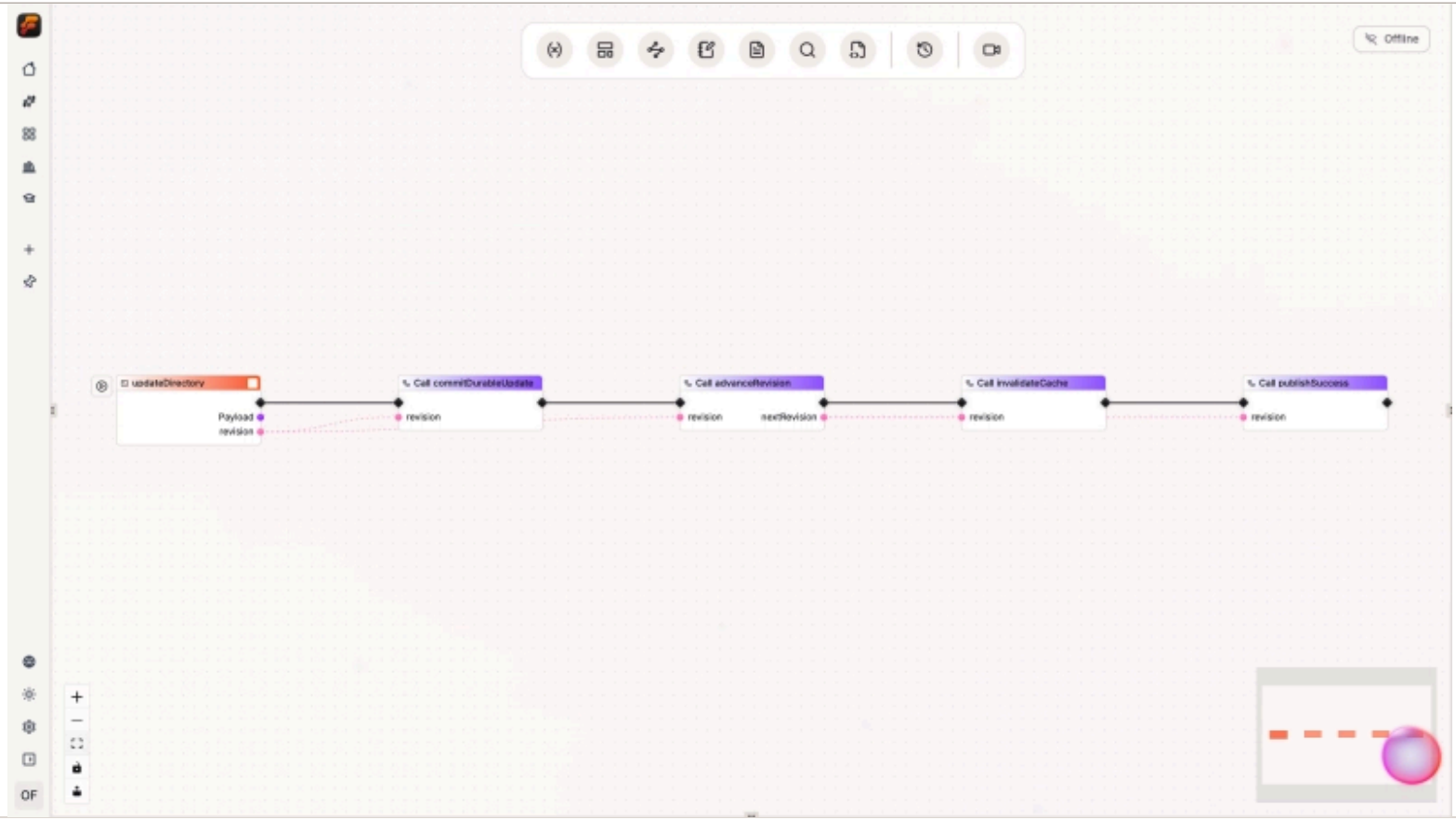
When the cache backend or entry is unavailable, the runtime warns and executes the function. Cache writes are best-effort. Concurrent identical misses are not combined, so the cache provides neither distributed locking nor exactly-once behavior.

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
}
```

Run this Event after a successful authoritative directory update, never after each read. The 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.

A lookup that missed before invalidation can finish afterward and write an old result back. Passing `directoryRevision` prevents new callers from using that key. Versioning the namespace also isolates it, though old entries remain until TTL or cleanup removes them.

Code, return names, package versions, model choice, and configuration do not clear cached results. Invalidate them when such a change alters the answer.

12.7 Use layers for readability, functions for sameness

An ordinary layer collapses part of one graph into a named nested view. Use one to reduce canvas clutter, label a section, or sketch a placeholder.

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. It preserves the inner nodes and typed boundary pins, then replaces the inline occurrence with a Call Function node.

Similar blocks may encode intentionally different policies. Extract them only when a change to one must be repeated in the others.

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 when a consumer needs `normalized`. The visible `if` makes `resolveSystem` structurally impure, but caching is safe because its inputs fully determine the side-effect-free result, including the directory revision.

The in-source mapping stands in for a governed production directory lookup. The fixture calls both Functions, adapts them through an agent Event, and invalidates the namespace after a durable update.

A function can unify shared logic, and an Event can expose it to independent callers. Add caching only when the runtime may safely skip the whole call.

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 correct Flow remains internal until someone or something can start it through a supported entry.

Here, the incident logic from the preceding chapters becomes an **Incident Desk App**. Its typed decision serves an App user, a schedule, and an authenticated REST caller. The interface is interactive, and each boundary can be governed, versioned, and traced.

Each surface gets an adapter suited to its caller while sharing one business function.

Release check: App Events bind compatible Flow entries to caller surfaces, configuration, variable overrides, execution location, and a Flow version. Roles govern Event execution and editing at App level; there is no per-Event ACL. Numbered versions work today. Weighted canary dispatch and some REST validation and evidence paths remain incomplete, as marked below.

13.1 An event node is an entry; an App Event is an exposure

Event names two distinct objects.

An **event node** lives on the Board. It starts 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 typed output pins. Chat and Mail Events provide specialized contracts.

An **App Event** configures how callers reach that entry. It selects the Flow, event node, and either Latest or a numbered Flow version. It also owns the interface, execution location where the surface offers a choice, surface settings, variable overrides, protected secret values, and the release information for that exposure.

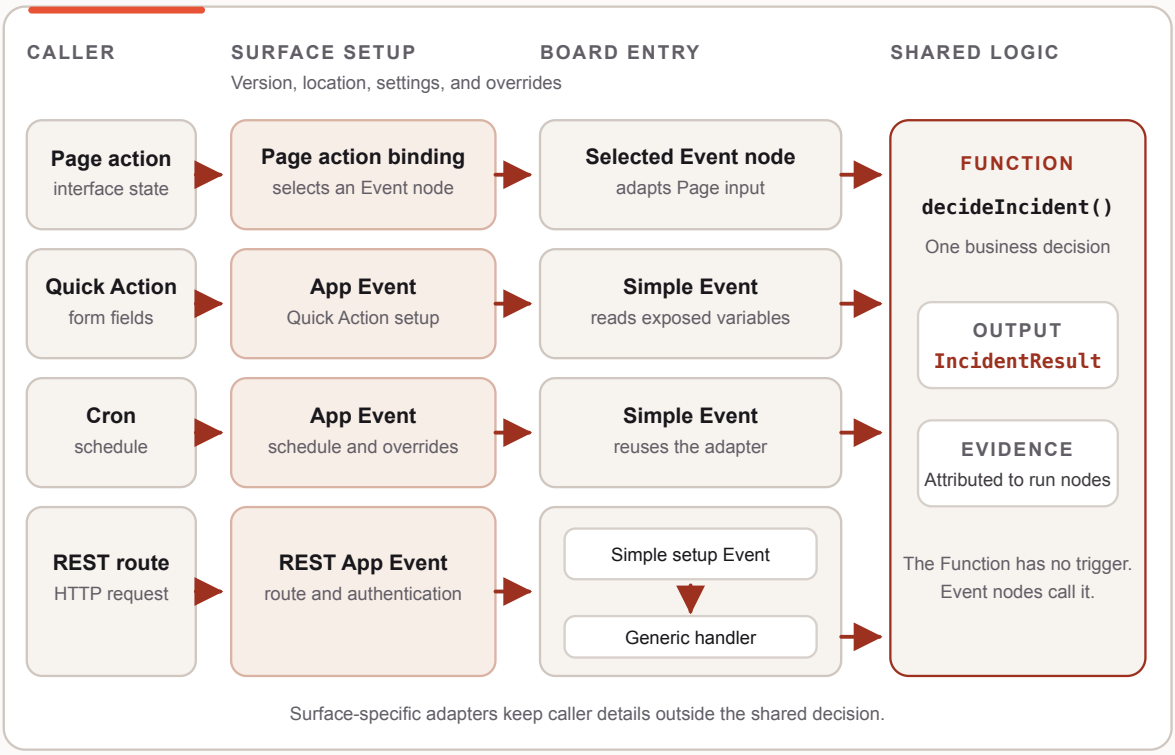
One event node may serve several compatible App Events with different schedules, settings, or variable overrides. A route change stays outside the business logic.

The event node defines the executable entry. The App Event defines its exposure.

An App Event cannot wrap every event node in every surface. Generic Events support Generic Form, API, and Deeplink setups. Simple Events support Quick Action, API, Cron, Daemon, Deeplink, REST, and MCP. Chat and Mail use specialized interfaces. The setup screen filters incompatible choices.

13.2 Keep one decision behind surface adapters

Our shared decision has a deliberately small result. The caller-specific path ends at an Event node adapter; the Function itself has no trigger.



Exposure stays outside the shared decision. Quick Action and Cron can reuse a Simple Event; REST setup registers a Generic handler. Every Event adapter calls the same typed Function.

```
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. Each interface adapts its input to the function and returns the same structured result.

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 the same Simple Event. With no interactive form, the schedule supplies variable overrides.

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 dispatcher fills those pins from the request; `return` publishes the REST response value.

`_client` carries trusted metadata from the authentication path, such as verified OAuth claims or a connected-App identity. The authentication path supplies this trusted metadata. The Flow may use it for finer authorization.

REST cannot register `decideIncident` directly because a Function has no trigger. Registration stores the ID of a concrete Event handler, so `triageRest` provides that adapter.

Keep shared logic in the Function and caller-specific behavior in the adapter. The visible adapter makes an interactive action, unattended schedule, and public endpoint easy to distinguish.

13.3 Try the interactive Page

The `/triage` Page collects an affected system, report, and business impact. Its button invokes the incident Event and renders the result.

The embedded React component is interactive. Change its values and press **Triage incident**. Its labelled local mock does not call Flow-Like or an external endpoint.

EMBEDDED REACT PROTOTYPE

Web · Remote

Incident Desk

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.

A production Flow-Like Page uses A2UI. Its component tree contains typed fields, a select, a button, and result cards. Components used during execution are marked event-relevant. The button action identifies the event node:

```
{
  "name": "workflow_event",
  "context": {
    "nodeId": "<triage-page-event-node-id>"
  }
}
```

The action context carries routing identity. On invocation, the client combines current Page elements and relevant values into the payload. The Flow reads them through A2UI catalog nodes, so no hidden JavaScript callback controls the button.

A UI App Event points to the Page and owns its route, such as `/triage`. Today the interface requires authentication. Anonymous Page hosting remains planned.

13.4 Quick Action and Cron reuse one entry

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 edits those fields and invokes `triageQuickAction`. A second App Event can target the same entry for a periodic job, overriding the values 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.

Flow authors choose which variables participate in configuration. Secret variables are handled separately and omitted from ordinary read responses. The REST API key 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 grants entry at the REST boundary; forwarding credentials to another system requires a separate decision.

13.5 A REST App Event is a setup program

A single API Event exposes one configured endpoint. For REST, the Flow builds a server configuration, registers handlers and authentication, publishes OpenAPI routes, then 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. It tells Register REST Function which handler to publish. Register one handler per remote route: 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 emits its configuration to the platform instead of binding a long-lived socket during setup. Saving the App Event runs the setup, checks that it reached REST Server, and persists the route and authentication registrations.

Setup is part of the release. A new REST Event rolls back when its first setup fails. If an update fails, traffic keeps using the last successful setup while the builder sees the error.

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 value becomes a `200 application/json` response. For more control, return an envelope with status, headers, content type, and body.

For named parameters, JSON object fields overwrite same-named query values. Reserved parameters provide the request, method, path, query, headers, raw body, and `_client` metadata. Keep the public contract small; accepting the whole request makes later compatibility harder.

Flow-Like publishes OpenAPI JSON and an interactive browser UI. The remote document describes routes, methods, and authentication, while request and response bodies remain open objects. The local socket imple-

mentation has richer pin-derived request schemas. The remote router does not yet enforce declared Flow types before execution.

The design target is stricter than the current release:

> **Doctrine:** Reject a request at the earliest boundary where its incompatibility is known, and record the rejection as execution evidence.

Malformed JSON, authentication failure, and unmatched routes are rejected before dispatch. Field values are not yet checked against the handler pin schema at the edge; a mismatch may fail later when a node evaluates its typed pin.

Pre-dispatch rejections do not appear as Runs. The status model has Pending, Running, Completed, Failed, Cancelled, and Timeout, with no Rejected state. Invalid JSON, failed authentication, and unmatched routes can return before a Run row exists. A type error that reaches execution can appear as a failed Run. The intended design records attributable rejection evidence without claiming that the request executed.

13.7 Identity, permission, and credentials are different boundaries

App members need **Execute Events** to invoke an Event. Builders need **Write Events** to create, activate, reconfigure, or repoint one. Owners and admins inherit these permissions; custom roles may receive them. App Events have no separate ACL.

After entry, user-context nodes expose the subject, role, permissions, and role attributes. The Flow can then ask whether the caller may triage payment incidents. The context model also supports custom attributes, though 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`

CONCERN	WHAT ANSWERS IT
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 unauthenticated access, API keys, static bearer tokens, Basic authentication, HMAC-SHA256, and OAuth/OIDC bearer validation. This exercise uses an API key. Leaving Register REST Auth disconnected creates an unauthenticated surface and requires deliberate review.

An API key does not identify a person. Verified OAuth claims can add subject, issuer, audience, scopes, and related metadata to `_client`. An Internal REST Event uses an approved App connection instead of the public router, while still enforcing authentication configured on the registration.

Interactive invocations may use the signed-in caller's credentials. Unattended and public sinks normally use credentials stored with the App Event or sink. Platform storage credentials have their own scope. Preserve those distinctions in the audit trail.

13.8 Hybrid applies to Boards

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; Studio can execute locally.
- A remote REST App Event is Remote by definition.

Hybrid lets builders work locally in Studio while the web experience runs on governed remote infrastructure. Each invocation still has one location. Its nodes may call remote systems, but the Flow run does not move between runtimes.

13.9 Pin the contract before other people depend on it

An App Event can follow **Latest** or target an immutable numbered Flow version. Latest exercises every saved Board change during authoring; production APIs should use a stable version.

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.

Callers cannot override the configured Flow version. REST registrations also keep a setup version and a pointer to the last successful setup. The Flow version freezes the Board snapshot; the setup version selects the published route configuration. Catalog implementations, Event configuration, data, credentials, and external systems retain their own lifetimes.

Weighted canary rollout is not yet a runtime guarantee. The model stores a canary target and weight; audited invocation paths still select the primary target. Treat canary settings as product intent until weighted routing is implemented 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 evidence item exposes a current gap: rejected requests should leave an attributable record just as failed execution does.

At that point, another team can operate Incident Desk without first learning its internal graph. When they inspect a run, its evidence still leads to the responsible block.

PART III

The Two-Way Contract

Open the machinery that keeps the visual Board and canonical FlowScript aligned through typed meaning and guarded changes.

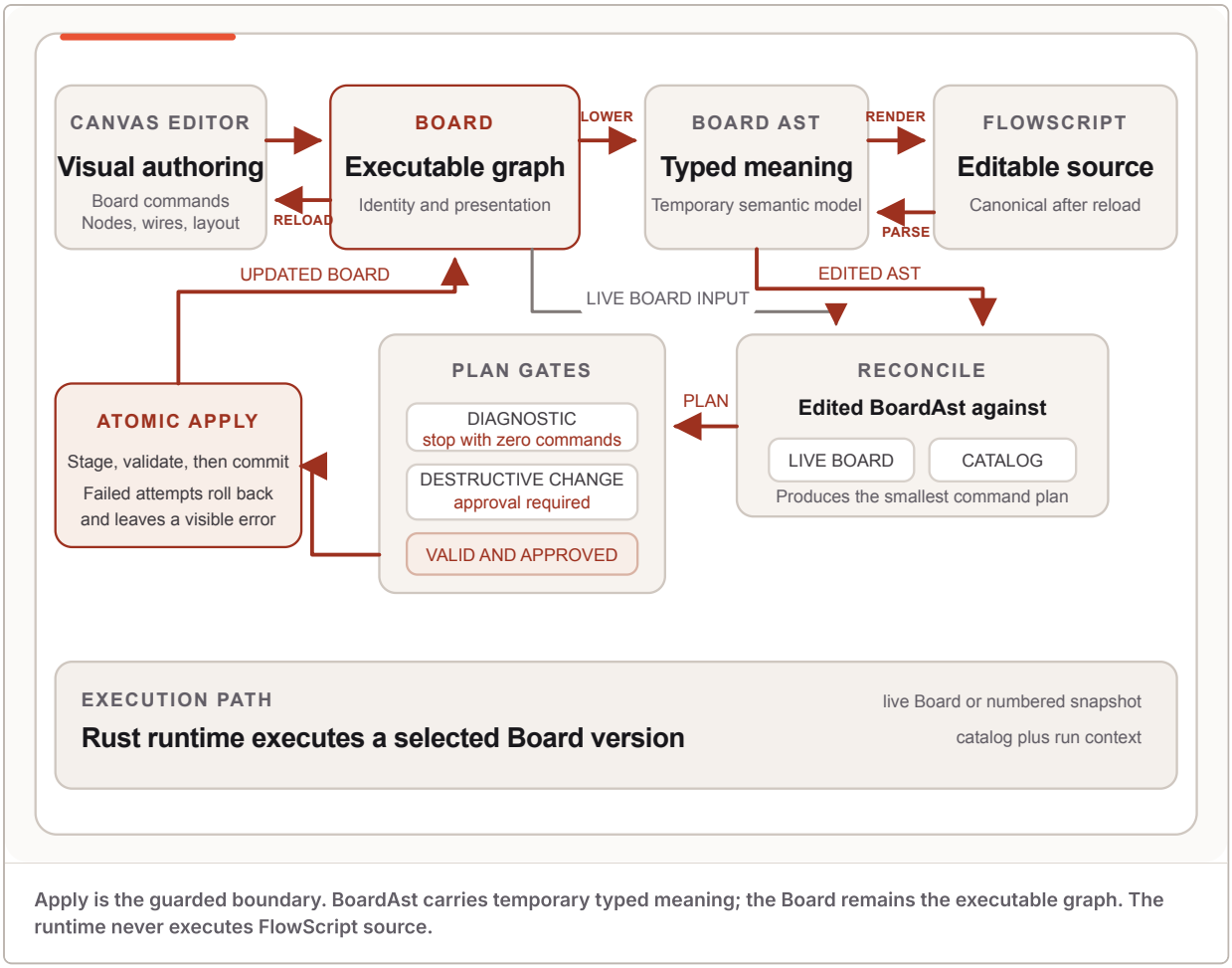
PART III · CHAPTER 14

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.

Editable text and an editable Board must stay in sync. A builder should be able to move one node without replacing the program; a developer should be able to change one literal without erasing the layout. Treating FlowScript as an export or rebuilding the graph after every text edit would break that contract.

Flow-Like uses a typed intermediate representation called `BoardAst`. It captures what the program means while the Board retains visual and runtime details.



Apply is the guarded boundary. BoardAst carries temporary typed meaning; the Board remains the executable graph. The runtime never executes FlowScript source.

The persisted Board remains executable. The AST is a temporary semantic contract. The canvas and FlowScript editors author the same Flow model.

Release check: FlowScript stays a draft until Apply. Studio can show a semantic command preview; Web receives authoritative diagnostics and the command outcome during Apply. Any diagnostic or unapproved destructive change blocks the batch, and failure rolls it back. Success reloads canonical source while preserving hidden secret values. FlowScript can rename an anchored Function in place while preserving the Function layer ID. It cannot yet change an existing anchored Function signature safely.

14.1 One program, three representations

Each representation owns different 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

REPRESENTATION	ITS JOB	INFORMATION IT OWNS
`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

Flow-Like does not store two programs and synchronize them later. Its contract is:

> *The canvas and FlowScript editors are equal authoring surfaces over one Flow model.*

Board edits change the next textual projection. Applied text edits become Board commands and change the next visual projection. The editors share executable meaning, while characters and pixels may differ.

A Board node needs an opaque identity so wires, logs, undo, and versions can keep referring to it. A programmer needs a readable name such as `productionStopped`. One serves storage; the other serves authoring.

14.2 Why Board JSON cannot serve as 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.

The clipboard contains useful transport JSON: serialized nodes, comments, cursor position, layers, variables, schema references, pin IDs, defaults, and connections. Flow-Like can mint new identities and reconnect the pasted selection.

At five hundred nodes, business decisions disappear inside storage identities, adjacency lists, coordinates, compatibility data, and UI metadata. Moving a node changes the JSON without changing the program. Any editor must manage graph bookkeeping before expressing a domain change. The AST separates those concerns.

Once text became an authoritative editing surface, especially for FlowPilot, readable graph output was insufficient. Direct text-to-Board conversion would tie language syntax to persistence and encourage whole-graph reconstruction. A typed semantic pivot lets both directions meet on meaning.

The first FlowScript implementation arrived with the FlowPilot update and introduced the AST, parser, renderer, Board lowering, and reconciliation together. The resulting division is:

- 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 starts with the live Board and catalog. A graph stores connections; a program needs scopes, expressions, statements, and order. The lowerer recovers those language structures.

It maps:

- Board variables to typed top-level declarations;
- schema-backed shapes to FlowScript interfaces;
- Function layers to typed `function` declarations;
- trigger nodes and their reachable execution paths to Event bodies;
- execution chains no trigger reaches to `detached` blocks;
- impure nodes on execution wires to ordered statements;
- pure data dependencies to nested expressions;

- named execution outputs to branches such as `onSuccess`, `execError`, or an `if` body; and
- node, variable, and layer identities to editing anchors.

A `detached` block represents an execution chain that no trigger reaches. FlowScript has no top-level statement position, and calling its ordinary root node an entry would hide its inputs. Nothing in the block runs; it reports existing Board content and cannot 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 stores the String Contains, variable reads, boolean operation, Branch, Set Variable, pins, and connections. Lowering recognizes the shape and chooses compact expression and control-flow syntax.

Visual helpers may disappear from the semantic projection. Lowering follows through reroutes and may flatten collapsed boundaries. Coordinates can order independent entries without becoming source syntax. The live Board retains this visual information while text is edited.

14.4 Rendering chooses one canonical spelling

Rendering `BoardAst` is deterministic. The renderer chooses document order, indentation, quotes, decorator order, spacing, parentheses, and declaration layout.

Canonical source makes comparisons smaller and removes debates over semicolons or quote style. After Apply, the system reloads and standardizes source from the Board.

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 expose precedence; quotes and semicolons are standardized. The semantic AST does not reproduce keystrokes.

Board lowering also chooses graph-derived sugar. Pure outputs may become nested expressions, and catalog metadata may turn a static call into a method. Imports appear only when they clarify calls without ambiguity.

If two namespaces expose `contains`, the lowerer keeps qualified calls:

```
const inReport = string::contains({
    string: report,
    substring: "production is on hold",
    ignoreCase: true
})
const inSystems = array::contains({ array: affectedSystems, value: systemId })
```

The qualified spelling remains until an import is unambiguous.

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 AST omits the secret value. An unchanged declaration carries no replacement value, so unrelated edits preserve the configured secret. FlowScript also rejects a nonempty initializer for a new secret.

This does not make downstream values safe to log or return, or imply that every sensitive pin and rendering path has been audited. Secret handling still depends on the node, execution boundary, and storage path.

14.5 Parsing turns a draft back into typed meaning

Typing does not mutate the Board. Flow-Like parses the draft for feedback and changes the Board only on a valid Apply. Studio can reconcile against a cloned Board for an authoritative command preview without persistence or undo mutation. Web performs that step during server Apply.

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?

Parsing cannot answer every program question. It recognizes `madeUp::operation({ value: 1 })` and named arguments without knowing whether the catalog contains the node or a compatible pin. Those checks need the live catalog and Board, so reconciliation owns them.

The editor has two 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 stays a draft with a specific problem to fix.

For experts: the small parser behind the editor

The parser is hand-written and 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 precedence and associativity. A shared nesting budget of 128 limits blocks, modules, expressions, and nested interface types in editor or API input.

Parsing stops at the first error and reports a line and column rather than a multi-error span set. Template-expression errors receive rebased source coordinates; later semantic diagnostics locate tokens on a best-effort basis. This supports a guarded editor, though it is not a finished compiler diagnostics system.

The formatter follows this invariant:

```
canonical = render(parse(authored))
render(parse(canonical)) == canonical
```

The stable second rendering takes priority over preserving authored style.

14.6 Reconciliation plans the smallest valid change

After parsing, the reconciler compares the edited AST with the Board. It resolves catalog calls, derives expected pins, checks types and shapes, accounts for dynamic pins, and matches existing entities through stable identities.

It produces a command plan for the persisted Board:

- 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 uses the same command vocabulary, though FlowScript reconciliation does not reposition existing nodes. New structures receive automatic placement; existing coordinates stay untouched.

This boundary preserves the Board. A literal edit can update one pin while leaving identities, coordinates, comments, connections, package fields, and visual choices alone. In Studio, a structural edit is visible in the preview before any command runs.

Studio's Apply preview groups semantic consequences, distinguishing configuration updates from structural or destructive edits. Web returns authoritative diagnostics and the command outcome during Apply without showing the plan beforehand. Deletions need separate approval; Chapter 15 covers the details.

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.

Apply preserves the accepted program and unrelated Board information, then restores canonical spelling.

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. Its identity is shortened here:

```
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 with a new string. The anchor identifies the existing Contains node. Because its `substring` input is unconnected and only its stored value changed, the plan contains one `UpdateNodePin`. The node is not recreated.

The change follows this path:

```
one source literal
    → one AST literal
    → one changed input pin
    → one Board command
    → the same node in the same place
```

For a structural edit, derive another signal 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"
}
```

With the standard catalog, the delta adds String Contains and Boolean OR operations, their literal settings, and data wires into the existing Branch. Other nodes retain their identity and placement. The exact plan depends on the installed catalog and live graph, so the product shows the real count.

The example does not add `customerFacing` to the anchored `decideIncident` Function. The reconciler rejects signature changes to established Function boundaries because it cannot yet migrate the layer and all callers safely. Create a new Function with the desired signature instead.

14.8 FlowScript leaves canvas presentation intact

A domain expert may place the severity branch beside its input, color its layer, add a Board comment, and route a connection around another cluster. FlowScript needs no keyword for these choices.

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.

Visual constructs need no equivalent source line. A reroute may disappear from FlowScript yet stay on the Board because no command touched it. Board comments are not lowered into FlowScript comments, and source comments cannot reliably create or replace them. A text-only rebuild cannot promise pixel-identical recovery.

> *FlowScript preserves supported program meaning and untouched Board identity. It does not preserve arbitrary author formatting or provide byte-for-byte or pixel-for-pixel serialization.*

An anchored Function rename emits `RenameLayer` and preserves the Function layer ID. Function signature migration remains unsupported in FlowScript. Whole-Board round trips also have known difficult cases involving duplicate handler names and cross-handler references. Report them with the Flow-Like version, a minimal Board, and the text before and after Apply.

14.9 The mental model to keep

Board edits retain graph identity and project readable source. A FlowScript edit proposes a semantic patch to that graph. The AST gives both paths the same typed meaning.

The opening figure is the working checklist. A Board lowers and renders to canonical source. An edited draft parses and reconciles against the live Board and catalog. After the plan passes its guards, Apply updates the Board and Studio reloads the 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.

PART III · CHAPTER 15

Identity, Anchors, and Safe Change

Learn how FlowScript anchors preserve Board identity, how reconciliation repairs deterministic drift, and how deletion guards, atomic Apply, and scoped editing protect existing work.

Rename the Incident Desk Event from `triageQuickAction` to `incidentTriageAction`. Its spelling changes. Its wires, position, App Event references, run evidence, and undo history must still point to the same Board node. A text edit that silently replaces that node would damage more than a label.

FlowScript carries durable graph identities in anchors such as `//@n:...`. The reconciler compares an anchor with the entity kind, containing scope, target, and boundary contract to decide whether an edit updates existing work or proposes a replacement. It repairs a change automatically when the evidence is decisive. Ambiguous plans stop. Recognized removals and moves to the Board root require approval.

Identity here means **Board entity identity**. Caller identity, authentication, and permissions belong to the App boundary described in Chapter 13.

Release check: Web and Desktop render editable source with anchors, apply guarded command batches, request separate approval for recognized deletions, and support selection-scoped editing. Studio/Desktop can show the authoritative command plan before Apply; Web performs the same reconciliation at Apply. Anchored Board-global variables, Events, and modules support in-place renames. Anchored Functions can be renamed or moved between modules without changing their layer IDs. Function signature changes remain rejected. Board command execution is rollback-backed and a successful Apply is atomic at the authoring layer. A successful mutating Apply is normally recorded as one local undo batch. Rollback, persistence, and history-bookkeeping failures remain visible.

15.1 Identity is not the same as spelling

The Incident Desk contains a String Contains node. Its FlowScript binding might be named `productionStopped`, `urgentPhraseFound`, or `customerImpactDetected`. Those names help a reader follow the rule. The Board node still needs an opaque ID so every other system can refer to the same operation after a rename.

For an author, the practical identity rule is:

> *Keep an entity's anchor when it occupies the same place in the program's meaning and continues to do the same job.*

The reconciler cannot infer purpose from prose. It tests properties the Board can verify: entity kind, containing module or Function, resolved target, and boundary contract. Event recovery also checks the handler scope. An anchored String Contains call cannot silently become an IMAP connection, and a call to one Function cannot keep its node identity while targeting another Function. Both edits fail those checks. An explicit move can change the containing module while preserving identity because the retained anchor makes the move itself part of the proposal.

Current behavior makes the boundary concrete:

EDIT	IDENTITY CONSEQUENCE
Rename an anchored Board variable	Updates the existing variable
Rename an anchored Event entry	Renames the existing entry node
Rename or reparent an anchored module	Updates the existing module layer
Rename an anchored Function	Renames the existing Function layer under its current ID
Move an anchored Function or Event to another module	Moves the existing entities under their current IDs
Change the catalog operation while keeping its node anchor	Blocks because the anchor and operation disagree
Copy Board content on the canvas	Mints new node, pin, and layer IDs and translates internal references
Move Board content	Retains IDs and changes its containing layer

The Function layer is the durable callable entity, and its anchor supplies the identity for an in-place rename. FlowScript emits `RenameLayer` when an anchored declaration receives a new name that is available in its module. Existing Call Function nodes continue to target the stable layer ID. Anchored Function signatures remain fixed because changing parameters or returns requires migrating every caller and boundary wire.

This distinction keeps familiar source syntax useful without asking names to carry more certainty than they can provide.

15.2 Read `//@n`, `//@v`, and `//@l` as identity

Editable FlowScript can include three anchor kinds. IDs are shortened here so the roles remain visible:

```
@description("Incident report supplied by the App")
let report = "Production is on hold"   //@v:incident-report

function decideIncident(report: string): (severity: string) {   //@l:decide-incident
    const productionStopped = report.contains({
        substring: "production is on hold",
        ignoreCase: true
    })   //@n:contains-report
    return productionStopped ? "SEV-1" : "SEV-2"
}

eventsSimple triageQuickAction() {   //@n:triage-entry
    const severity = decideIncident(report)   //@n:call-decision
    log::info({ message: severity, toast: false })   //@n:record-severity
}
```

The prefixes map directly to Board entities:

ANCHOR	ENTITY
`//@n:`	A node, including an Event entry or Function call node
`//@v:`	A variable, either Board-global or Function-local
`//@l:`	A Function or module layer

They are trailing comments because the current source contract needs identity to travel with the line that owns it. The parser recognizes only the known anchor forms. An anchor on its own line is normally left unattached rather than assigned to the preceding entity. The deliberate ex-

ception is an unambiguous `//@n:` before a branch arm. The renderer emits the anchor kind appropriate to each parsed entity in its canonical trailing position.

Keeping identity inside the document lets external editors, copy operations, diffs, scoped renders, source-to-Board navigation, and statement-level merges carry the same reference.

Leave anchors in place during ordinary edits. Studio can dim them without deleting them. When copying source to create another entity, leave the anchor on the original statement. Remove it from the pasted copy before Apply, then rename the copy as needed so declarations remain unique. Do not invent anchors. An ordinary anchor ID that is absent from the current Board is reported as unavailable and ignored, so it cannot force an association with live work. The entity then follows ordinary unanchored reconciliation. An absent Event anchor stays attached for the specialized recovery described in Section 15.4. One anchor assigned to two distinct entities produces a diagnostic before the reconciler looks it up or derives any Board commands.

During a merge, combine competing edits to an anchored statement into one surviving statement. If both versions must remain, only the continuation keeps the anchor. The other version becomes a new entity and receives a new identity during Apply.

15.3 Reconciliation preserves the existing Board

An anchor gives the reconciler an exact starting point. It can compare the edited entity with the live node, variable, or layer and emit the smallest command that represents the semantic change.

Rename the Incident Desk Event while preserving its anchor:

```
- eventsSimple triageQuickAction() {   //@n:triage-entry
+ eventsSimple incidentTriageAction() {   //@n:triage-entry
```

The current plan contains one `RenameNode` for `triage-entry`. The Event body, entry pins, wires, coordinates, and App Event references retain their existing identity. Renaming an anchored Function or module simi-

larly uses one `RenameLayer`. The Function keeps its layer ID, body, boundary pins, cache settings, and existing call targets. Moving an anchored Function to another module uses `MoveToLayer` without recreating it.

Written callers in the same revision must use the Function's final name. Planning retires its old spelling. Qualified calls likewise use the module's final path after a rename or move.

Changing a literal is smaller still:

```
- substring: "production is on hold",
+ substring: "customer-facing outage",
```

With `//@n:contains-report` intact, the reconciler can emit one `UpdateNodePin` for the existing Contains node. Chapter 14 followed that exact path.

Reconciliation leaves untouched Board facts where they already live. New source still creates new entities: the planner assigns temporary references, adds the required nodes and layers, connects their pins, and places them without re-laying out existing work. The same command vocabulary makes a one-character update and a new branch comparable during review.

15.4 Corrections automate only proven repairs

A **correction** is a deterministic repair that does not require the system to choose among plausible meanings. A correction can accompany a valid plan. A **diagnostic** says the proposal cannot be represented safely or unambiguously. When any diagnostic exists, Apply executes zero commands.

Duplicate-anchor preflight runs against the submitted source before any Board lookup. One anchor assigned to two distinct entities is ambiguous even when the ID is absent from this Board. A multi-output Event header and its immediate first arm-routing Branch may repeat the same node anchor because both source forms represent one Event entry. Any distinct reuse stops before the reconciler derives commands.

After preflight, an anchor ID absent from the current Board is unavailable for an ordinary node, variable, Function, or module. The reconciler reports a correction, ignores that ID, and handles the entity as unanchored. Ordinary resolution may create a new entity or reuse one unique compatible live target. It cannot use an absent ID to preserve identity.

A live Function or module layer named by an anchored declaration is reserved for that declaration throughout the revision. If the same source revision renames `legacyHelper` to `currentHelper` and also declares a new unanchored `legacyHelper`, the new declaration receives another layer. Source order does not let it capture the layer being renamed. The same rule applies when a Function moves out of a module and its old location receives a new Function with the same name.

Availability does not prove compatibility. An anchor that resolves on the current Board remains authoritative. If its live entity has the wrong operation, layer kind, Function target, or boundary contract, reconciliation fails closed. A variable anchor that exists in another Function or in the Board globals is also incompatible with the declaration's scope. It remains live and is not cleared as unavailable. Unanchored resolution stops when several live entities are equally plausible, including duplicate Function layers or same-named variables in one scope.

Event recovery shows the boundary. For an absent Event anchor, the reconciler collects unclaimed, registered entries that fit the written type, name or alias, scope, and parameter contract. If several entries fit, exactly one entry that still drives the first still-live execution body node can settle the choice. One remaining candidate is re-anchored.

With zero or several compatible candidates after that check, recovery does not guess. It uses the normal entry-creation path and assigns a new Board identity. An exact catalog type stays exact. An alias-only header uses the standard Simple or Generic fallback according to its declared parameters. Valid Event metadata is still required, but an absent anchor and an empty copied handler do not by themselves block recreation.

Absent Event anchors keep this specialized recovery path instead of being cleared during the unavailable-anchor pass. That distinction lets Event recovery consider kind, scope, and parameter contract before it chooses a unique entry or creates a replacement.

Unavailable anchors do not turn a whole-Board replacement draft into an import. In replacement mode, omitting existing target content can still propose deletions. Use additive or explicitly scoped authoring for a partial cross-Board paste, then review the resulting command plan.

Other blocking cases include:

- one anchor attached to several distinct entities;
- a live anchor whose entity kind or operation disagrees with the written entity;
- an ordinary unanchored declaration with several compatible live targets;
- a Function call anchor that now names a different Function;
- an anchored Function boundary with changed parameters or returns; and
- an unresolved or incompatible catalog call.

Apply results carry corrections separately from diagnostics. A correction causes the editor to reload canonical FlowScript even when no Board mutation was needed, so the repaired identity appears in the next source projection. The current editor does not explain the correction in a separate message.

Re-anchor only when the evidence identifies one compatible live entity. When a new entity is a valid representation of the source, recreation avoids choosing among live candidates. Return a diagnostic when neither a safe update nor a valid creation represents the proposal.

15.5 Deletions require explicit intent

Anchors also make absence meaningful. If an anchored, text-visible node disappears from a whole-Board or in-scope draft, the reconciler can plan its removal. Recognized removals and moves from a module to the Board root require a separate approval.

Use the Incident Desk for a safety drill. Start with the anchored Contains statement:

```
const productionStopped = report.contains({
    substring: "production is on hold",
    ignoreCase: true
})   //@n:contains-report
```

Now remove only its anchor:

```
const productionStopped = report.contains({
    substring: "production is on hold",
    ignoreCase: true
})
```

The text still looks equivalent to a reader. The identity claim has changed. The current reconciler treats the unanchored call as a new Contains node and sees the old `contains-report` node as absent. Its plan adds a replacement, rewires the expression, and removes the existing node. The removal makes the plan destructive, so the first Apply executes no commands.

Studio/Desktop can expose that replacement plan before Apply. Web reaches the authoritative guard during Apply and shows its destructive summary. The confirmation names or identifies the recognized Board items and offers two paths:

1. Keep everything, restore `//@n:contains-report`, and apply the intended in-place edit.
2. Apply with deletions when replacing the node and its identity is deliberate.

Ordinary editor approval triggers a fresh reconciliation and authorizes the recognized destructive commands in its result. It does not pause for a second review if that result differs from the preview. Refresh before approving when the Board may have changed.

Deleting the entire anchored statement produces the same need for approval without the replacement. Removing a variable also enters the destructive gate. Moving content from a module to the Board root is guarded because a missing module wrapper could otherwise dissolve a namespace. Studio/Desktop's proactive classifier currently misses that relocation, so the authoritative guard catches it during Apply.

The gate follows command families rather than claiming to recognize every risky semantic change. Disconnecting a wire is not automatically classified as deletion, for example, so review the complete plan when it is available. Function and module retirement also remain current gaps: omitting a declaration can remove visible body nodes without removing its layer. Retire those boundaries from the Board until FlowScript emits a complete layer-removal plan.

AI-authored text follows the same rule as a human edit or an external tool. A truncated response, partial paste, or narrow model context receives no extra authority to remove unseen work.

15.6 A successful command batch becomes one undo unit

FlowScript Apply treats a document as one proposed program change. Apply follows this sequence:

1. Parse the complete draft and reconcile it against the live Board and catalog.
2. Return zero executed commands when a diagnostic exists.
3. Return zero executed commands when a recognized destructive plan lacks approval.
4. Execute the validated setup and remaining phases, rolling back earlier commands if a later phase cannot be built, executed, or validated.
5. On Web, persist only after the complete plan succeeds, then return the executed command batch.

For a successful mutating Apply, Web and Desktop normally push the returned nonempty batch into local history as one entry. One press of Undo submits the whole batch; the Board applies its inverse commands in reverse order. Redo validates and executes the batch forward again. A structural source edit therefore remains one author action even when it required several node, pin, wire, and layer commands.

Atomicity covers the Board authoring mutation. It does not execute the Flow or roll back external side effects from a later run. Rollback can fail, and local history bookkeeping can fail after a Board has been persisted. Both failures remain visible. When revision stamps are available, the clients clear stale history rather than replaying an inverse batch against a different revision.

Desktop applies and saves locally, then delivers the same command receipt for shared Boards. Its shared-board path can restore a pre-Apply snapshot after an Apply or save failure. An offline save failure is a narrower edge: the error remains visible, but the in-memory Board can be ahead of storage and requires explicit recovery before further editing. The current path does not restore it automatically.

The useful promise is observable and bounded: a successful command batch lands as one authoring change, and a failure remains visible with recovery information.

15.7 Scoped editing protects the unseen remainder

A large Flow does not need to become one large editing task. On a live Board version, select one or more nodes on the canvas, open the context menu, and choose **Edit selection as FlowScript**. This entry is available through the shared Web and Desktop Board interface.

Flow-Like expands that selection into a coherent source scope:

- the complete Event, Function, or detached chain that contains a selected node;
- every Function those sections reference, followed transitively;
- all Board variables and interfaces as shared context; and
- `use` declarations recomputed for the resulting document.

The render also returns the anchors of the included top-level sections. Apply sends those `scope_anchors` back with the edited source. The reconciler still reads the live Board and catalog, but its Event and Function node-removal diff covers only the declared scope. An Event or Function omitted because it was never rendered cannot be interpreted as deleted.

An entity removed inside the visible scope still requires deletion approval. Board variables remain visible as shared context, so removing one is also an in-scope destructive edit.

The editor banner reports how many sections are in scope and states that out-of-scope content is untouched. Leaving the scope returns to the current whole-Board source.

Scoped editing gives people and tools a smaller reasoning surface with an explicit completeness boundary. It is valuable for a focused human change, a generated patch, and a review that should not carry the entire Flow into working memory.

Preserve anchors during ordinary editing. Remove one only when replacement is deliberate and the destructive review is acceptable.

The next chapter turns this identity model into editor behavior: navigation, completion, diagnostics, formatting, previews, and the language tools that connect a cursor back to the Board.

FlowBook · A Developer's Guide to Flow-Like

Open edition · 2026 · © 2026 Flow-Like

book.flow-like.com