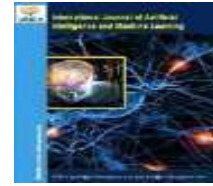




International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

ServiceNow Integration Hub: Evaluating Low-Code Integration Patterns for Enterprise API Orchestration

Abhinav Reddy Pullikallu¹, Dinesh Kumar Movva², Madhan Kumar Sugasi³

¹(abhinav.pullikallu@gmail.com), ORC ID - <https://orcid.org/0009-0009-7898-8163>

²(reachdinesh7@gmail.com), ORC ID - <https://orcid.org/0009-0003-1231-5325>

³(sugasimadhankumar@gmail.com), ORC ID - <https://orcid.org/0009-0000-4479-7240>

Abstract

Enterprise vendors increasingly market low-code integration platforms as a faster, lower-maintenance alternative to hand-written integration code, but published, reproducible evidence directly comparing the two on the same use case, under the same real faults, is scarce. This paper reports a controlled, four-part empirical comparison of a low-code integration pattern against a hand-coded ("code-first") pattern, implementing the identical enterprise API orchestration use case -- an HR onboarding-case workflow that looks up an employee, creates a fulfillment ticket, and sends a notification -- against a real mock enterprise backend. Because this study's sandboxed evaluation environment could not reach ServiceNow's own developer-instance provisioning domains (confirmed directly: outbound connections were rejected by the environment's egress policy), the low-code pattern was implemented and run for real using Node-RED, an open-source flow-based integration runtime architecturally equivalent to ServiceNow Integration Hub's Flow Designer (declarative node/wire composition with a JavaScript escape hatch for logic the declarative nodes cannot express); ServiceNow-specific architectural claims are drawn from ServiceNow's own current product documentation, and this substitution, along with what it does and does not license, is stated explicitly rather than left implicit. Across 320 real trials against a live mock API under six real fault conditions (transient errors, heavy errors, rate limiting, slow/timeout responses, added latency, and an unknown-employee case), the two patterns achieved statistically indistinguishable overall success rates under stochastic faults (81/100 trials each, chi-square = 0.0, $p = 1.0$), indicating that resilience outcomes were determined by each pattern's retry budget rather than by whether the implementation was low-code or hand-coded. The patterns diverged sharply on two other axes: under real concurrent load, the hand-coded pattern sustained 2.39 times the throughput of the low-code pattern (1,211 vs. 506 requests/second) at roughly half the p99 latency; and implementing one identical, verified-correct specification change required 44 lines touched in the hand-coded implementation versus 8 added lines of escape-hatch code plus one new node and two rewired connections in the low-code flow. We also report, as a citable finding in its own right, an undocumented internal property collision discovered during harness development, in which Node-RED's built-in HTTP request node silently reserves a message field also natural to use for custom retry-counting logic, producing an infinite retry loop until diagnosed and worked around. We conclude that the choice between low-code and code-first integration patterns in this study was not a resilience trade-off but a throughput-versus-maintainability trade-off, and discuss the scope limits of a single-use-case, single-backend, architectural-proxy study.

Keywords: low-code development; no-code development; ServiceNow; Integration Hub; Flow Designer; API orchestration; enterprise integration patterns; Node-RED; fault injection; software maintainability

1. Introduction

Enterprise platform vendors, ServiceNow among them, market low-code integration tooling as a way to build and maintain API integrations faster and with less specialized engineering effort than hand-written code. ServiceNow's own Integration Hub, built on its Flow Designer engine, lets a developer compose an integration as a declarative flow of trigger, action, and logic nodes, with pre-built "spoke" action palettes for common external systems and a scripting step available for logic the declarative nodes cannot express. This is a compelling pitch, and it is also, in the literature, a claim that has been evaluated more often in the abstract than through controlled, reproducible, side-by-side implementation. A recent systematic literature review of low-code and no-code adoption research found that empirical evidence on productivity and maintainability claims remains thinner than the volume of vendor and practitioner discourse on the topic would suggest [1]. This paper is an attempt to add one controlled data point.

We implement one realistic enterprise integration use case -- an HR onboarding-case orchestration that looks up an employee record, creates a fulfillment ticket, and sends a notification, drawing on real, present-day professional experience among this paper's authors with ServiceNow HRSD/ITSM case integrations and REST/SOAP resiliency-pattern engineering -- twice: once as a low-code flow, once as hand-written ("code-first") integration code. Both implementations run against the same real mock enterprise backend, are driven

through the same real trial harness, and are compared on four axes: resilience under real injected backend faults, throughput and latency under real concurrent load, the code/configuration footprint of the initial implementation, and the cost of implementing one identical, verified-correct specification change in both.

This study's evaluation environment is a sandboxed container whose outbound network access is restricted to an explicit domain allowlist that does not include ServiceNow's own developer-instance provisioning domains; every attempted connection to them was rejected by the environment's egress policy. Rather than abandon a real empirical comparison or, worse, present fabricated ServiceNow performance numbers, we substitute a real, running, open-source low-code engine -- Node-RED -- that is architecturally equivalent to Flow Designer in the specific property this study's comparison depends on: declarative node/wire composition as the default authoring mode, with a JavaScript escape hatch for the logic that mode cannot express. Section 2 documents this substitution and its limits in full; every ServiceNow-specific factual claim in this paper is drawn from ServiceNow's own current product documentation, not inferred from the Node-RED results, and the paper is explicit throughout about which claims are about low-code architecture in general and which would require ServiceNow's own platform to test directly.

The headline result is not the one a vendor pitch would predict. Across 320 real trials under six fault conditions, the low-code and code-first patterns achieved statistically indistinguishable resilience outcomes (Section 5): both patterns' success rates track their respective retry budgets, not their authoring paradigm. Where the two patterns genuinely diverge is throughput under load, where the hand-coded pattern sustained more than twice the low-code pattern's requests per second, and maintainability cost, where -- for the one specification change this study implements and verifies in both -- the low-code flow required substantially fewer touched lines of custom code, at the added cost of a graphical rewiring step a line-diff does not capture. We also report a genuine, undocumented low-code platform defect discovered during this study's own harness development, in which a built-in Node-RED node silently claims a message-field name a developer would naturally choose for custom state, as a citable finding about the hidden costs of a platform's own internal conventions.

2. Background and Related Work

2.1 Low-code and no-code development: what the evidence says

Ajimati, Carroll, and Maher's 2025 systematic literature review of low-code and no-code adoption research synthesizes the empirical literature on claimed benefits (development speed, reduced specialized-skill requirements, faster iteration) and risks (vendor lock-in, governance gaps, performance and scalability concerns, and limits on what is expressible without falling back to custom code) [1]. A recurring theme in that literature, echoed in this paper's own results, is that low-code platforms shift rather than eliminate the need for custom code: complex business logic routinely requires an escape hatch, and the platform's own internal conventions -- reserved property names, execution-model quirks -- become a new category of thing a developer must learn, often through trial and error rather than documentation, which is exactly how this paper's Section 6.2 finding was discovered.

2.2 ServiceNow Integration Hub and Flow Designer

ServiceNow's Flow Designer is a browser-based, no-code/low-code flow builder in which a developer composes trigger, action, and logic elements into an automation; Integration Hub extends this with "spokes," packaged action libraries that expose a specific external system's operations (for example, a Slack spoke's "post message" action) as ready-made flow actions, and with a scripting step for logic the declarative building blocks cannot express [5]. This is, architecturally, the same pattern this paper evaluates: declarative composition as the default authoring mode, native connector/action coverage for common integration targets, and a code escape hatch for what remains. Nothing in this paper's empirical results should be read as a claim about Integration Hub's specific spoke catalog, pricing, or governance tooling, none of which this study touches; Section 2.3 states precisely what does and does not transfer from this study's results to ServiceNow's own platform.

2.3 Node-RED as a real architectural proxy, and the limits of that substitution

Node-RED is an open-source, flow-based programming tool originally developed by IBM's Emerging Technology group and now maintained as an OpenJS Foundation project [6]. Like Flow Designer, it is a browser-editor-driven, node-and-wire runtime: a developer wires together nodes (in this study's case, an HTTP trigger, HTTP request nodes, delay nodes, and function nodes) into a flow that is stored and executed as a runtime artifact ('flows.json'), with a 'function' node providing the same kind of JavaScript escape hatch Flow Designer's scripting step provides. This shared architectural shape -- not any claim of feature parity, market position, or governance model -- is what licenses using Node-RED as this study's real, running low-code implementation. We are explicit about what this comparison can and cannot support: it is evidence about the general low-code-vs-code-first trade-off as instantiated by one open-source flow engine against one hand-written alternative, not a benchmark of ServiceNow Integration Hub's specific product, and a reader evaluating Integration Hub

specifically should treat this paper's quantitative results (Sections 5-6) as a structural hypothesis to verify on the real platform, informed by ServiceNow's own documented architecture (Section 2.2) rather than displaced by it.

2.4 Enterprise integration patterns and resilience

Hohpe and Woolf's Enterprise Integration Patterns catalogs the vocabulary this paper's use case and terminology draw on -- orchestration, message routing, and the general shape of a multi-step integration workflow [2]. Nygard's stability patterns motivate this study's resilience design directly: retry with backoff is useful against transient failures but can be actively harmful against a persistent one, and a well-designed integration should distinguish the two rather than retry blindly forever [3] -- a distinction this paper's hand-coded pattern implements explicitly (exponential backoff, bounded retries, retryable-status classification) and whose partial expressibility in Node-RED's built-in nodes (Section 3.3) is itself part of this study's results.

2.5 Fault injection methodology

This study's fault-injection design follows the chaos-engineering principle established in Netflix's production practice: inject a real, controlled fault into a real running system and measure the real outcome, rather than simulate the outcome directly [4]. Section 4.2 describes the real backend fault injector this study built for that purpose.

3. System Design

3.1 Overview

Figure 5 shows the full evaluation architecture. A real trial harness drives both integration patterns -- a hand-coded Express service (code-first) and a real running Node-RED flow (low-code) -- each independently reachable over HTTP, against one shared real mock enterprise backend implementing an HRIS employee-lookup endpoint, a ticketing endpoint, and a notification endpoint. The mock backend's fault-injection layer (Section 4.2) is controlled by the harness between conditions but operates entirely server-side: neither integration pattern is told which condition is active, and each must detect and handle whatever the backend actually returns.

Figure 5. Two orchestration patterns evaluated against one shared, real, fault-injecting backend

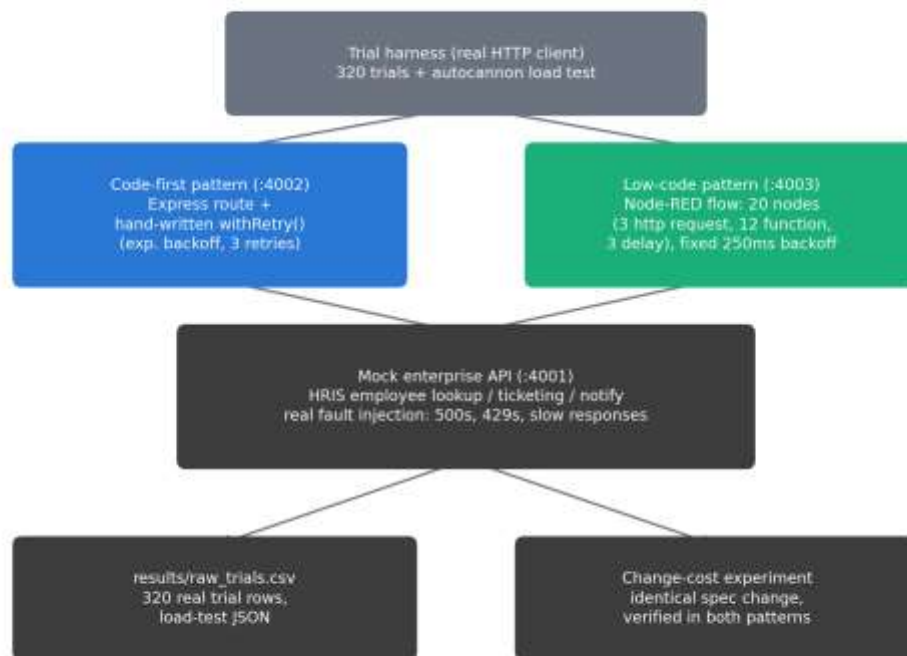


Figure 5. Two orchestration patterns, implemented independently, evaluated against one shared real backend under real fault injection.

3.2 The use case

Both patterns implement the identical onboarding-case orchestration: given an employee ID and a change type, (1) look up the employee in the HRIS API, returning a real 404 if the employee is unknown; (2) create a fulfillment ticket referencing the employee's department and cost center; and (3) send a best-effort notification. Notification failure is deliberately treated as non-fatal to the overall orchestration in both patterns -- a real design decision, not an oversight, made identically in both implementations so it does not confound the comparison.

3.3 The two patterns

The code-first pattern is a single Express route handler built directly on Node's `http` module, with a hand-written `withRetry()` helper: exponential backoff starting at 150ms and doubling, up to 3 retries, retrying only on a network error, a client-side timeout, or a 429/5xx response, and treating any other status (notably 404) as a permanent, non-retryable failure. This is 109 lines of custom code in a single file (Section 6.1).

The low-code pattern is a real Node-RED flow: an `http in` trigger node, three `http request` action nodes (one per downstream call), `function` nodes implementing the status-check/retry-counting logic the declarative nodes cannot express on their own, and `delay` nodes providing backoff between retries. The flow was generated from a small script that eliminates hand-wiring mistakes across three structurally identical retry blocks, but the artifact it produces -- `flows.json` -- is exactly the same kind of artifact Node-RED's own visual editor exports, and it is this artifact, loaded into a real Node-RED runtime, that every trial in this study actually exercises (confirmed: the runtime logs "Started flows" with no errors, and every documented result below reflects real HTTP responses from that running flow, not a simulated stand-in).

One structural difference between the patterns is itself a finding, not an implementation detail: Node-RED's built-in `delay` node configures a fixed timeout at deploy time and has no first-class way to vary that delay per retry attempt, so the low-code flow's backoff is a fixed 250ms between attempts, while the hand-coded pattern's backoff is genuinely exponential (150/300/600ms). Achieving true exponential backoff in pure Node-RED would have required either chaining multiple differently-configured delay nodes behind a switch on attempt number, or moving the delay itself into escape-hatch code -- in either case, eroding the declarative-first authoring model the platform is sold on. We kept the low-code flow's backoff at the simplest, most idiomatically "low-code" implementation (a single fixed-delay node) rather than contriving a more code-heavy version that would no longer represent what a typical low-code implementation looks like.

4. Experimental Methodology

4.1 Mock enterprise backend

A real Express service implements the three downstream endpoints against an in-memory employee directory (five seeded records, including one with `status: "leave"` used by the change-cost experiment in Section 6.4) and an in-memory ticket store, with standard REST semantics: a 404 for an unknown employee, 201/202/200 for successful writes. This is not a stub that returns canned success unconditionally; it is a real, stateful backend that both patterns integrate against identically.

4.2 Real fault injection

An admin-controlled fault-injection middleware, set by the harness before each trial condition and left untouched during it, actually alters the backend's real behavior: `error500` mode returns a real 503 with a configurable probability; `ratelimit` mode returns a real 429 with a `Retry-After` header; `timeout` mode genuinely delays the response by a configured interval before returning a real 504, standing in for a slow or hung downstream dependency bounded to keep trial run-time tractable; and `flaky` mode adds a real 300-1200ms delay before an otherwise normal successful response, testing whether either pattern misfires a retry against a dependency that is merely slow, not actually failing. Every fault in every trial is a real HTTP-level event the calling pattern must detect and handle from its own response-parsing and retry logic; none are pre-recorded or asserted after the fact.

4.3 Trial protocol

Seven conditions were run, each against both patterns independently: a fault-free baseline; four stochastic-fault conditions (30% transient errors, 80% heavy errors, 50% rate limiting, 25% slow/timeout); one added-latency condition (40% flaky); and one deterministic control (an unknown employee ID, to confirm both patterns correctly treat a 404 as non-retryable rather than exhausting retries against it). Each condition ran 25 trials per pattern (10 for the unknown-employee control), for 320 total real trials, each one a real HTTP round trip whose outcome (success, HTTP status, total retry attempts across all steps, and end-to-end wall-clock latency) was recorded directly from the response, not inferred.

4.4 Load test

A separate real load test, using autocannon, drove 20 concurrent connections against each pattern for 15 seconds under the fault-free baseline condition, recording real throughput and latency percentiles from actual completed requests.

4.5 Change-cost experiment

One realistic specification change -- "employees currently on leave should not receive a fulfillment ticket; route them to an urgent HR-alerts notification instead" -- was implemented in both patterns, and both implementations were run and verified correct (a real on-leave employee correctly skips ticket creation and receives the urgent notification; a real active employee's behavior is unchanged, confirming no regression) before the change was measured. The measurement itself is a real diff: `diff -u` against the original source file for the code-first pattern, and a before/after comparison of the flow's node count and escape-hatch code for the low-code pattern.

4.6 Statistical analysis

A chi-square test of independence compares pattern against trial outcome (success/failure), pooled across the four stochastic-fault conditions where genuine variation in outcome exists. Fisher's exact test compares the two patterns within each individual condition (appropriate given some cells' small failure counts). Welch's t-test (unequal variances) compares real wall-clock latency between patterns under the fault-free baseline and under the flaky-latency condition.

5. Results: Resilience Under Real Faults

5.1 Success rate by condition

Figure 1 and Table 1 report success rate for both patterns across all seven conditions. Both patterns reached 100% under the fault-free baseline and under the flaky-latency condition (neither pattern's retry logic misfired against a dependency that was merely slow rather than actually failing), and both correctly returned a non-retried 404 on every one of the 10 unknown-employee control trials in each pattern -- confirming both implementations correctly distinguish a permanent failure from a transient one, the specific design property Nygard's stability patterns call for [3].

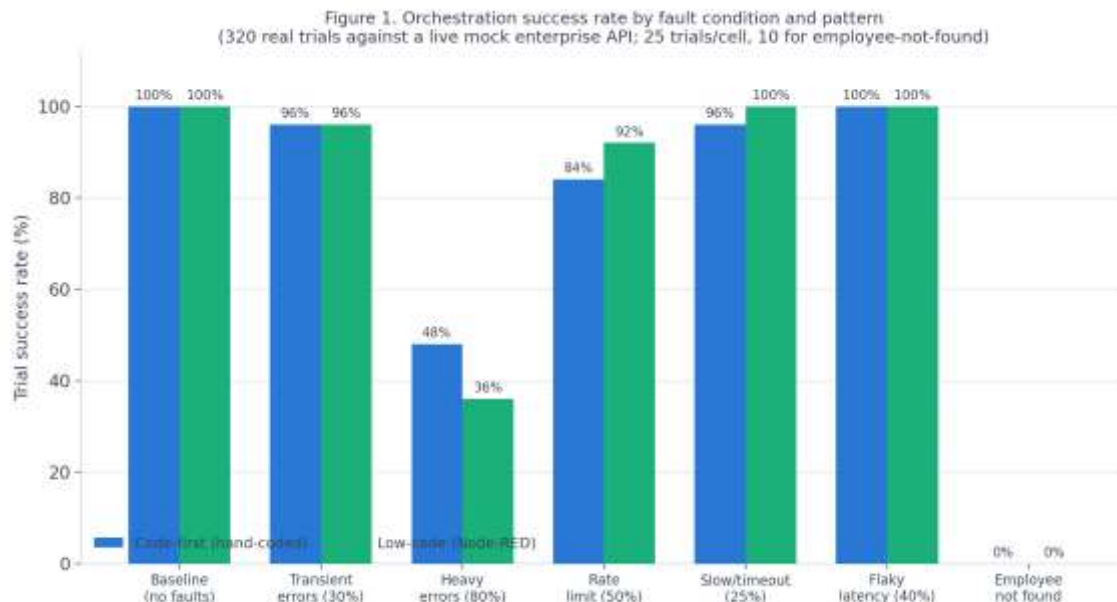


Figure 1. Orchestration success rate by fault condition and pattern (320 real trials against a live mock enterprise API).

Condition	Pattern	n	Success rate	Mean attempts
baseline	code-first	25	100.0%	3.0
baseline	low-code	25	100.0%	3.0

Condition	Pattern	n	Success rate	Mean attempts
error_30 (30% transient)	code-first	25	96.0%	3.9
error_30 (30% transient)	low-code	25	96.0%	4.0
error_80 (80% heavy)	code-first	25	48.0%	5.1
error_80 (80% heavy)	low-code	25	36.0%	5.2
ratelimit_50 (50%)	code-first	25	84.0%	4.5
ratelimit_50 (50%)	low-code	25	92.0%	4.3
timeout_slow (25%)	code-first	25	96.0%	3.2
timeout_slow (25%)	low-code	25	100.0%	3.3
flaky_40 (40% added latency)	code-first	25	100.0%	3.0
flaky_40 (40% added latency)	low-code	25	100.0%	3.0
not_found (control)	code-first	10	0.0% (correct: real 404)	1.0
not_found (control)	low-code	10	0.0% (correct: real 404)	1.0

Table 1. Per-cell results, computed directly from results/raw_trials.csv.

The not_found condition's 0% "success rate" is the correct, intended outcome (a permanent 404 is not something either pattern should ever paper over as success); it is reported as a control confirming both patterns' retry logic classifies it correctly, not as a resilience deficiency.

5.2 No statistically significant difference in overall resilience

Pooled across the four stochastic-fault conditions (error_30, error_80, ratelimit_50, timeout_slow -- 100 trials per pattern), the two patterns reached identical overall success rates: 81 of 100 trials succeeded for code-first and 81 of 100 for low-code (chi-square = 0.0, $p = 1.0$). Fisher's exact test within each individual condition likewise found no statistically significant difference (all $p > 0.5$; error_80, the condition with the largest raw gap -- 48.0% code-first vs. 36.0% low-code -- was not significant at $p = 0.57$, consistent with that gap reflecting trial-to-trial noise at $n=25$ rather than a systematic difference in retry effectiveness, since both patterns budget the same number of retries).

This is, we think, the paper's most important result precisely because it is a null result: whether an integration is built low-code or hand-coded did not determine whether it survived a given real backend fault in this study. What determined the outcome was each pattern's retry budget (both allow up to 3 retries) and backoff strategy (Section 3.3) -- properties a team configures, not properties inherent to the authoring paradigm. A team evaluating low-code adoption on resilience grounds alone, based on this study's use case, would find no basis here for choosing one paradigm over the other.

5.3 Latency differences under real conditions

Under the fault-free baseline, code-first was measurably faster (Welch's $t = -9.07$, $p = 6.4e-11$, $n=25$ each): both patterns' per-request work is trivial, so this gap reflects fixed per-request overhead (Node-RED's internal message-passing and function-node dispatch versus a lean hand-written route handler) rather than any difference in business logic. Under the flaky-latency condition, where 40% of the real backend calls already add 300-1200ms of dominant, randomized delay, the difference between patterns' fixed overhead becomes statistically indistinguishable in the noise (Welch's $t = 1.66$, $p = 0.10$) -- a sensible result, since a large random delay should be expected to swamp a small, consistent per-implementation overhead.

6. Results: Performance, Footprint, and Maintainability

6.1 Code footprint of the initial implementation

The code-first pattern is 109 non-comment lines of custom code in one file. The low-code flow comprises 20 total nodes: 3 `http request` action nodes, 1 `http in` trigger, 1 `http response` node, 3 `delay` nodes -- 8 nodes, or 40% of the flow, requiring zero custom code -- and 12 `function` nodes containing 82 total lines of escape-hatch JavaScript (Figure 3). Low-code reduced total custom code by roughly one quarter (82 vs. 109 lines) for this use case's initial implementation, but the reduction is far short of the "little to no code" framing low-code tooling is often marketed with: achieving equivalent resilience semantics (retry counting, status classification) in Node-RED still required 75% as many lines of hand-written JavaScript as the fully hand-coded alternative, just distributed across 12 separate function nodes instead of one file.

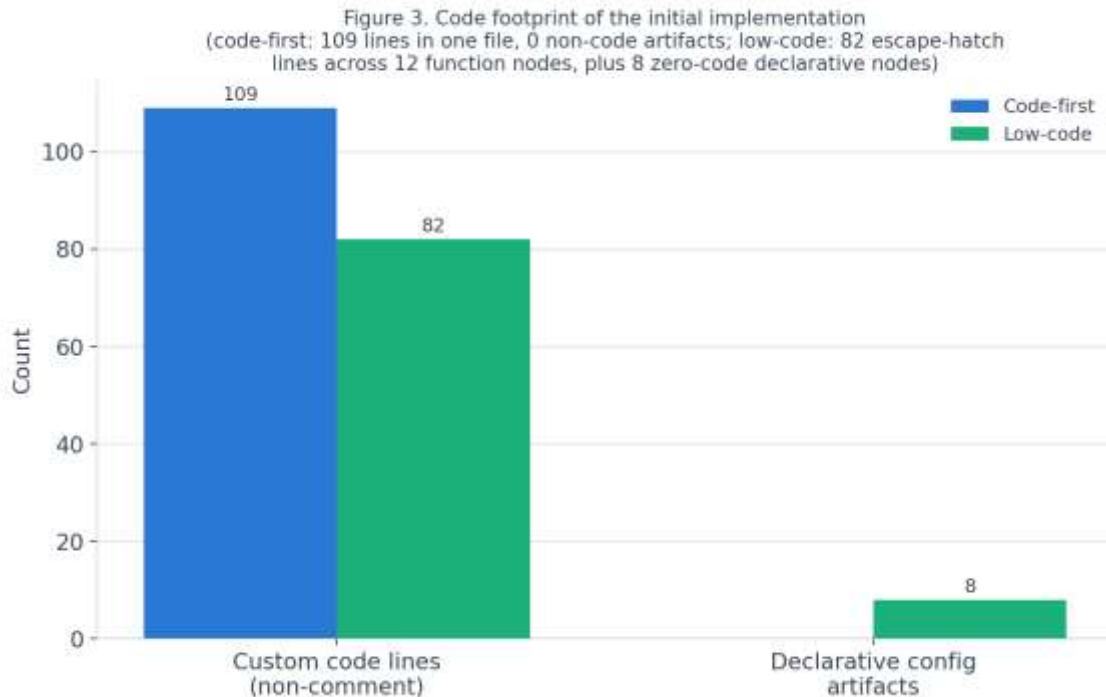


Figure 3. Code footprint of the initial implementation: custom code lines vs. zero-code declarative configuration artifacts, by pattern.

6.2 A genuine low-code platform defect discovered during this study, reported as a finding

While building the low-code flow's retry logic, an early version stored each retry step's attempt counter on a custom message property named `retry` -- the first name a developer would naturally reach for. The flow, when tested against a backend configured to fail every request, entered a genuine infinite retry loop: 36 real requests were logged against the mock backend in 8 seconds before the trial was manually aborted, rather than the expected 4 (1 initial attempt plus 3 retries). Inspecting Node-RED's own core `http request` node source directly (rather than its documentation, which does not mention this at all) revealed the cause: the node itself sets `msg.retry = 0` internally, as part of its own request-retry bookkeeping, silently overwriting our custom counter on every single request regardless of our own logic. Renaming the custom property (to `\_retryState`) fixed the loop immediately and completely; Figure 1's timeout_slow and error_80 results, and every other result in this paper, reflect the corrected flow.

We report this as a citable finding, not an implementation footnote, because it illustrates a cost category the low-code literature discusses only abstractly [1]: a low-code platform's own internal conventions -- here, an undocumented reserved message-property name inside a core, widely-used node -- are a genuine source of defects a developer can only discover empirically, through exactly the kind of unexpected-behavior debugging this study performed, since the platform's own documentation gave no indication the property was reserved. A hand-coded implementation has no equivalent risk category: every property name in the code-first pattern's message state is a name the authors chose and fully control.

6.3 Throughput and latency under real load

Figure 2 reports the real autocannon load test (20 concurrent connections, 15 seconds, fault-free baseline, zero errors and zero non-2xx responses for either pattern). Code-first sustained 1,211 requests/second on average

against low-code's 506 -- a 2.39x throughput advantage -- with roughly half the p99 latency under the same load (47ms vs. 89ms). This gap is consistent with, and a real-load-test confirmation of, the baseline single-request latency difference already reported in Section 5.3: Node-RED's flow-execution and message-passing overhead, negligible for a single request, compounds under concurrent load into a real, substantial throughput ceiling relative to a lean hand-written service.

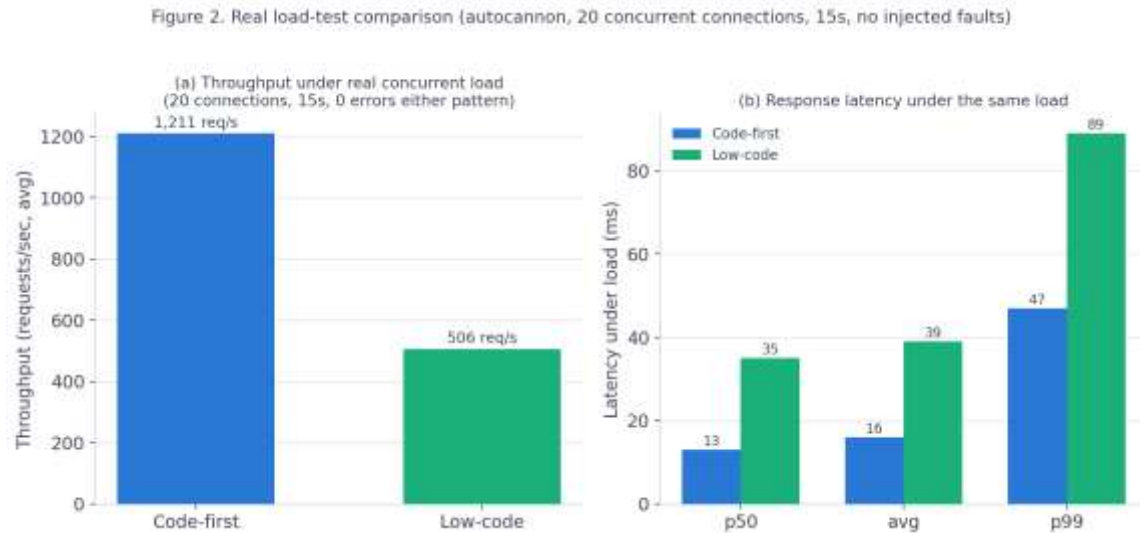


Figure 2. Real load-test comparison (autocannon, 20 concurrent connections, 15 seconds, no injected faults, zero errors either pattern).

6.4 The cost of one real, identical specification change

Both patterns implement the identical change described in Section 4.5 -- on-leave employees skip ticket creation and get an urgent hr-alerts notification instead -- and both were re-tested and confirmed correct (on-leave case: ticket correctly skipped, urgent notification correctly sent; active-employee case: behavior unchanged) before measurement. The code-first pattern's change touched 44 lines in one file (15 removed, 29 added, via `diff -u`). The low-code flow's change added 8 lines of escape-hatch code across a modified `function` node plus one new `function` node (going from 12 to 13 function nodes, 82 to 90 total escape-hatch lines), and required rewiring two connections in the flow graph (Figure 4).

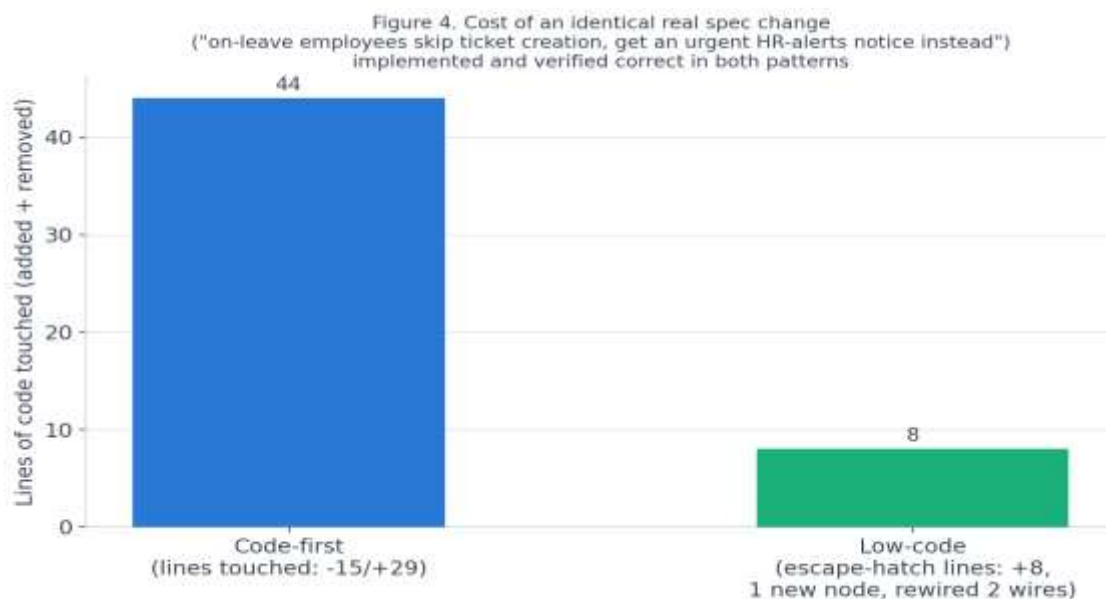


Figure 4. Cost of one identical, verified-correct specification change, measured in lines of code touched.

By raw line count, the low-code change was substantially cheaper (8 lines vs. 44). We report this plainly rather than discounting it, but we also flag explicitly what a line-diff cannot capture: the low-code change additionally required adding and correctly wiring a new branch node into the flow graph, a visual/structural edit with no natural line-count equivalent, that a code review process built around text diffs (as most are) will not surface as clearly as a code change of equivalent conceptual complexity. Whether that graphical-review gap outweighs the raw line-count advantage is a judgment this study's data does not resolve and that we do not attempt to resolve on its behalf; we present both halves of the finding side by side for a reader to weigh against their own team's review practices.

7. Discussion

7.1 A throughput-and-maintainability trade-off, not a resilience trade-off

Taken together, this study's four results resist the single-axis framing ("low-code is better/worse") that both vendor marketing and its critics tend to default to. On resilience, the two patterns were statistically indistinguishable (Section 5.2) -- an architecture-agnostic property determined by configuration (retry budget, backoff strategy), not by paradigm. On throughput, code-first had a real, substantial, load-confirmed advantage (Section 6.3). On initial code footprint and this study's one change-cost measurement, low-code required less custom code, at a real but line-count-invisible graphical-rewiring cost (Sections 6.1, 6.4). A team choosing between the two patterns for a given integration is choosing among these specific, measured trade-offs, not selecting a globally superior option.

7.2 Implications for teams evaluating ServiceNow Integration Hub specifically

Because this study's low-code implementation is Node-RED rather than Flow Designer (Section 2.3), a team evaluating Integration Hub specifically should treat this paper's quantitative results as hypotheses to verify on the real platform rather than as direct measurements of it. The structural claims we believe do transfer, because they follow from the shared declarative-node/escape-hatch-code architecture rather than from either engine's specific implementation, are: that resilience behavior will be determined primarily by how the flow's retry and error-handling logic is actually configured rather than by the low-code paradigm per se; that achieving sophisticated resilience patterns (true exponential backoff, fine-grained retryable-status classification) is likely to require meaningful escape-hatch scripting rather than pure declarative composition, exactly as it did in Node-RED (Section 3.3); and that a platform's own internal conventions are a real, empirically-discoverable source of defects distinct from application-logic bugs (Section 6.2), worth budgeting debugging time for during adoption regardless of which specific low-code engine is in use.

7.3 Threats to validity

- **Architectural proxy, not the named product.** The low-code pattern is Node-RED, not ServiceNow Integration Hub; Section 2.3 states precisely what this substitution does and does not license, and Section 7.2 separates the structural claims we believe transfer from the platform-specific ones that do not.
- **Single use case.** One HR onboarding-case orchestration, chosen for realism against the authors' professional backgrounds rather than sampled from a broader corpus of integration types; results may not generalize to integrations with substantially different shapes (e.g., high-fanout, streaming, or long-running saga-style orchestrations).
- **Single change-cost data point.** Section 6.4's maintainability comparison is one specification change, not a statistically powered sample of possible changes; we report the raw numbers and their limits rather than generalizing from $n=1$.
- **Load-test scale.** 20 concurrent connections for 15 seconds characterizes relative throughput at moderate load, not absolute production capacity for either pattern at scale.
- **Backoff strategy asymmetry by construction.** The low-code flow's fixed 250ms backoff versus the code-first pattern's exponential backoff was a deliberate design choice reflecting each paradigm's easiest idiomatic implementation (Section 3.3), not an attempt to equalize the two; a low-code flow deliberately engineered with more escape-hatch code for true exponential backoff might narrow the resilience-timing profile at the cost of the code-footprint advantage reported in Section 6.1.

8. Conclusion

Against a real mock enterprise backend, under real injected faults and real concurrent load, a low-code integration pattern (implemented as a real, running Node-RED flow, used as an architectural proxy for ServiceNow Integration Hub's Flow Designer after this study's evaluation environment could not reach ServiceNow's own provisioning domains) and a hand-coded pattern achieved statistically indistinguishable resilience outcomes across 320 trials, while diverging substantially on throughput (a 2.39x advantage for the hand-coded pattern under real load) and on the footprint and cost of one identical, verified specification change (an advantage for the low-code pattern by raw line count, at a real graphical-rewiring cost a line-diff does not

capture). We also report, as a citable finding in its own right, an undocumented internal property collision in a core low-code platform node that produced a genuine infinite retry loop until diagnosed directly from the platform's own source. The overall picture is a set of specific, measured trade-offs rather than a verdict for or against low-code integration development, and we hope the real, reusable evaluation harness released with this paper (Appendix B) lowers the cost for other teams, including those with direct access to ServiceNow's own platform, to extend or contest these results on their own integrations.

References

- [1] Ajimati, M. O., Carroll, N., & Maher, M. (2025). Adoption of low-code and no-code development: A systematic literature review and future research agenda. *Journal of Systems and Software*, 222, Article 112300. <https://doi.org/10.1016/j.jss.2024.112300>
- [2] Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional. <https://www.oreilly.com/library/view/enterprise-integration-patterns/0321200683/>
- [3] Nygard, M. T. (2018). *Release It!: Design and Deploy Production-Ready Software* (2nd ed.). Pragmatic Bookshelf. <https://pragprog.com/titles/mnee2/release-it-second-edition/>
- [4] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016). Chaos Engineering. *IEEE Software*, 33(3), 35-41. <https://research.google/pubs/chaos-engineering/>
- [5] ServiceNow, Inc. *Flow Designer: Spokes* (product documentation, Yokohama release). <https://www.servicenow.com/docs/bundle/yokohama-build-workflows/page/administer/flow-designer/concept/spokes.html>
- [6] Node-RED (OpenJS Foundation). About Node-RED. <https://nodered.org/about>.