Sample report. Northwind Pay is a fictional company. Every finding, screenshot, and transcript here is illustrative, built to show what a real Friction Audit deliverable contains.

DEVEX FRICTION AUDIT

Friction Audit Report

Northwind Pay, Inc.

A two-week evaluation of the Northwind Pay developer platform across seven dimensions, including agentic usability. Six findings, ranked by impact against effort, each with evidence and a fix.

PRODUCT	Usage-based billing infrastructure
STAGE	Series B
ACTIVE DEVELOPER ACCOUNTS	2,400
SURFACE AUDITED	REST v3, SDKs, docs, MCP beta
DELIVERED	14 August 2026
PREPARED BY	Mark Hazlewood, MarkedUp Consulting

01

Executive summary

Northwind Pay has a well-built API with a documentation problem that has quietly become a revenue problem.

3 of 3

agent runs produced duplicate charges on retry

Finding 1

11 min 40 s

measured time to first successful charge

Against a documented promise of five minutes. Finding 2

2 of 3

agent runs selected the wrong endpoint to charge a customer

Finding 3

Over two weeks I worked through your platform the way a new integrator would. I built three small applications against the v3 API, read the documentation cold, broke things on purpose to see what the errors said, and ran an AI coding agent against the same surface with no assistance. I found nineteen friction points. Six are written up here.

The most expensive one is not a bug. Your charge endpoint honors an Idempotency-Key header that appears nowhere in your documentation and nowhere in either SDK. Human integrators who have used Stripe send the header out of habit and are protected. Agents, and anyone reading only your docs, do not send it. When a request times out and gets retried, the customer is charged twice. Your team told me at kickoff that 18% of support tickets last quarter were tagged duplicate charge. That number has a documentation-shaped cause and a one-page fix.

The second is your quickstart. It promises five minutes and it cannot deliver five minutes, because step 3 requires a publicly reachable HTTPS endpoint before any charge will resolve. A developer evaluating you on a Tuesday afternoon has to install a tunneling tool and register a URL before they see anything succeed. That is where your 31% signup-to-first-call conversion is going.

The third is agent readiness, and it is the one with the longest tail. Your MCP server is in beta and its tool descriptions are one line each. Given a plain task with no hints, an agent picked the wrong endpoint in two of three runs. Agent traffic against billing APIs is growing and it arrives with ordinary credentials on ordinary endpoints, so you cannot currently measure it. You can, however, design for it, and almost none of your competitors have started.

None of the three requires re-architecting anything. The first two are documentation and defaults. The third is a schema pass on eleven tool definitions. I have ordered everything by impact against effort, and the four highest-value items in this report total about three engineer-weeks.

02

Scorecard

Seven dimensions, each scored against the published rubric. There is no composite score on purpose. Averaging these into one number would invite an argument about weighting and give you nothing to act on.

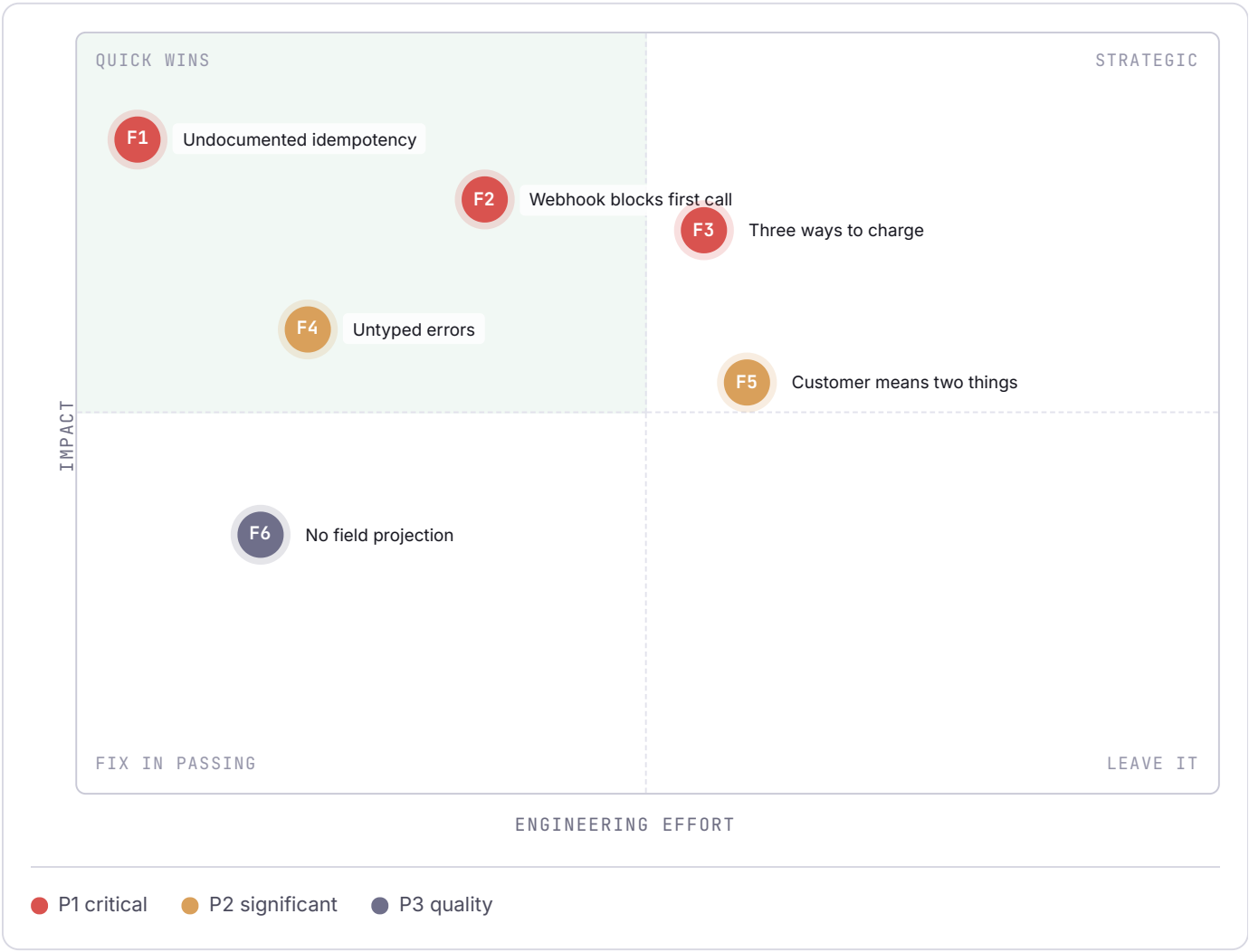
01	**Time to Hello World**	A tunneling tool stands between signup and first success. Measured 11m40s against a documented five minutes.	1
02	**Agentic Usability**	Mean of ten criteria. Undeclared idempotency and one-line tool descriptions are the two that cost you most.	0.9
03	**Error Experience**	Validation failures return an untyped string with no field pointer across the whole v3 surface.	1
04	**Documentation Quality**	Well organised and mostly accurate. Every sample I ran worked. Gaps are omissions, not errors.	2
05	**Conceptual Model Alignment**	Customer means two different things depending on the subsystem, and nothing in the docs says so.	1
06	**Feedback Loops & Debugging**	The dashboard request log is genuinely good. Webhook delivery history is the missing half.	2
07	**API Design Consistency**	v3 is internally consistent. The surviving v1 endpoints are the exception, and they are still load-bearing.	2

Scored 0 to 3 against the published rubric. 0 absent, 1 partial, 2 adequate, 3 strong.

03

Impact against effort

Every finding plotted by what it is worth to you against what it costs your team. Effort is estimated from the outside, so treat it as a starting point for your own sizing rather than as a number to plan against.



04

Methodology

Everything in this report came from using the platform, not from reading it. Here is what that involved, so you can judge the findings on how they were produced.

01 **Built three integrations**

A metered API billing flow, a one-off checkout, and a monthly invoice reconciliation script. Node and Python. All against test mode, starting from a fresh account with no prior context.

02 **Read the documentation cold**

Every page under Guides and API reference, in the order a new integrator would meet them. I ran every code sample as written rather than assuming it worked. All of them did.

03 **Provoked 24 failure modes**

Missing fields, wrong types, expired keys, insufficient scopes, malformed IDs, rate limit overage, oversized payloads, and mid-request timeouts. Every response recorded verbatim.

04 **Ran an agent against the surface, cold**

Six realistic tasks phrased as intent rather than as endpoints, given to Claude Sonnet 4.5 with access to your MCP server and your public docs. No hints, no corrections. Each task run three times, because one failure is an anecdote and three is a finding.

05 **Reproduced every finding by hand**

Each agent failure was confirmed with a plain curl call. That separates defects in your surface from quirks of a particular model, and it is what makes these findings survive a skeptical engineer.

Nineteen friction points came out of that. Six are written up here, chosen for consequence rather than for count. The remaining thirteen are listed in the appendix.

05

Findings

Six findings, ordered by severity. Each one carries the evidence it came from, what it costs you, and a fix specific enough to hand to an engineer.

P1 F1 Agentic Usability AU6 Safe retry

Idempotency works but is undocumented, so retries create duplicate charges

Impact **High** Effort **Low**

WHAT I FOUND

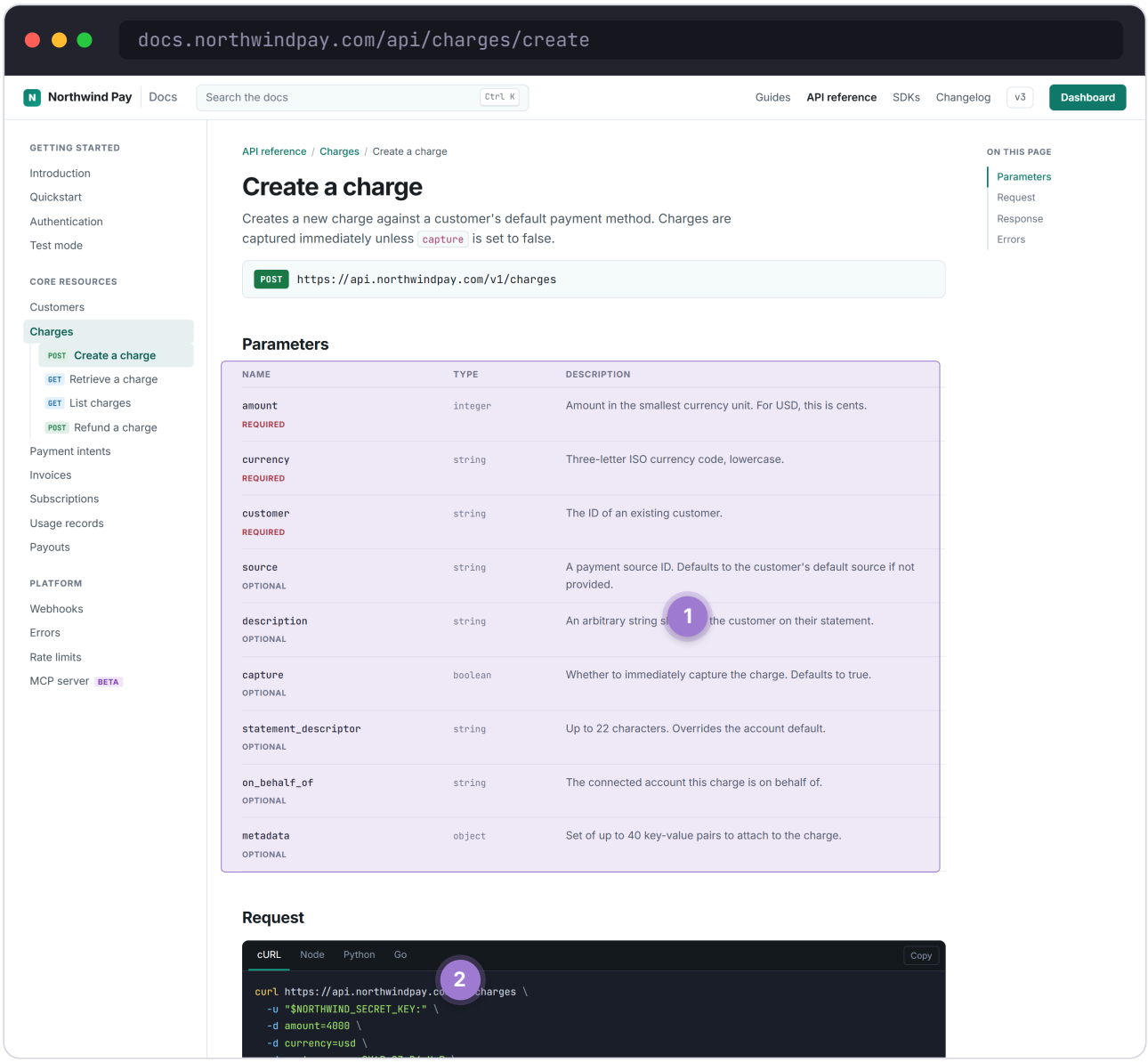
POST /v1/charges accepts an Idempotency-Key header and handles it correctly. Send the same key twice and you get the same charge object back, with no second charge created. I confirmed this by hand across four repeat calls.

The header appears nowhere in the reference page for the endpoint. It is not in the parameters table, not in the request sample, and not in the errors section. Neither northwind-node nor northwind-python exposes it as an option, so an SDK user cannot send it without dropping to a raw HTTP call.

The behaviour therefore only protects integrators who guessed it was there. Most of them guessed because they have integrated Stripe before, which is a strange thing for your reliability to depend on.

The failure is worst under timeout, which is exactly when retries happen. Your gateway returns 504 on requests that exceed 30 seconds, and the charge frequently completes on your side after the 504 has already been sent. A caller that retries then charges the customer a second time.

EVIDENCE



1 Nine parameters documented. Idempotency-Key is not one of them.

2 The request sample omits it too, so nobody copying this is protected.

EXHIBIT 1.1 The parameters table for POST /v1/charges. Nine parameters are documented. The Idempotency-Key header, which the endpoint honors, is not among them, and is absent from the request sample below the table.

EXHIBIT 1.2 An agent charging a customer, with the gateway timing out on the first attempt. Three of three runs produced the same outcome. All three charges settled.

RECOMMENDED

201 Created

POST /v1/charges

REQUEST

```
POST /v1/charges HTTP/1.1
Authorization: Bearer sk_live_...
Idempotency-Key: 8f14e45f-ea4d-4b90-9c1b-2a7d6f0e91cc
Content-Type: application/json

{ "amount": 4000, "currency": "usd", "customer": "cus_9KtRm2ZqB4vHnP" }
```

Documented in the parameters table, and generated by the SDK when the caller omits it.

RESPONSE

```
Idempotency-Replayed: true
```

Tells the caller this is the original charge, not a second one.

```
{
  "id": "ch_3PqKw1Jm9bXcQe",
  "amount": 4000,
  "status": "succeeded",
  "idempotency_key": "8f14e45f-ea4d-4b90-9c1b-2a7d6f0e91cc"
}
```

Echoing the key lets a caller reconcile a retry against the original.

WHY IT COSTS YOU

Your team put duplicate-charge tickets at 18% of support volume last quarter. Each one costs a support cycle, a refund, and a customer who now checks their statement.

The cost grows rather than holds steady. Retry-on-timeout is the default behaviour of every agent framework, and agents retry far more readily than humans do. As agent-written integrations increase, this defect scales with them.

This is also the cheapest finding in the report to fix. The capability already works. What is missing is a paragraph of documentation and an SDK parameter.

THE FIX

1. Document the header on the charges reference page, in the parameters table, and in every code sample. Do the same for every other write endpoint that already supports it.
2. Add an idempotencyKey option to both SDKs, and have the SDKs generate a UUID automatically when the caller does not supply one. Automatic generation is the change that fixes this for the callers who will never read the docs, agents included.
3. Return the key in the response body as idempotency_key so a caller can confirm which request a charge belongs to.
4. Add an Idempotency-Replayed: true response header on replayed requests, so callers can distinguish a replay from a fresh charge.
5. Longer term, reject unkeyed writes on the charges endpoint behind a version flag. That converts a silent failure into a loud one.

P1 F2 Time to Hello World

The quickstart requires a public webhook endpoint before any charge resolves

Impact **High** Effort **Medium**

WHAT I FOUND

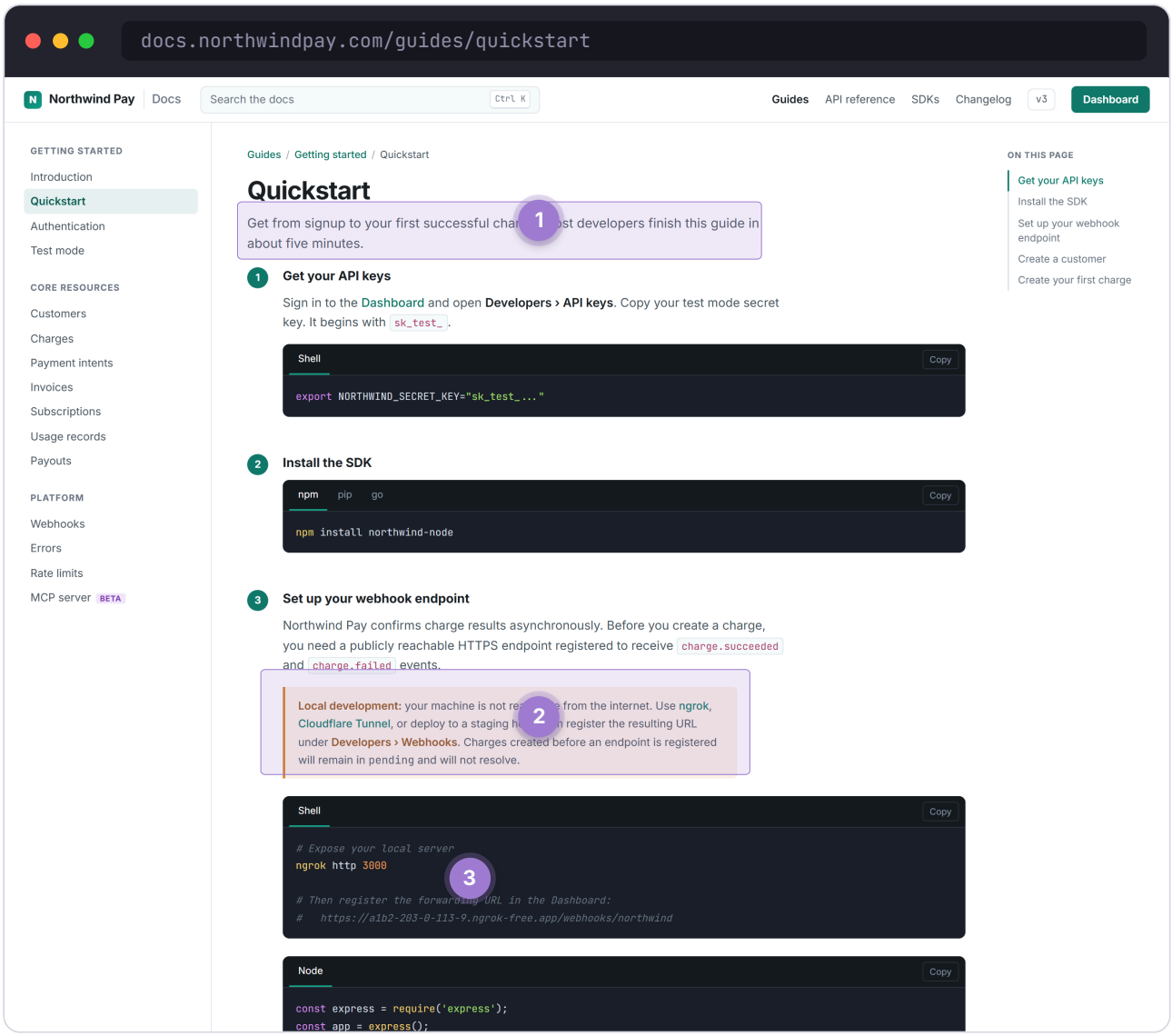
Your quickstart promises five minutes. I ran it cold on a fresh account and reached my first settled charge in 11 minutes 40 seconds. The gap is entirely step 3.

Step 3 asks for a publicly reachable HTTPS endpoint registered to receive charge.succeeded before any charge will resolve. On a laptop that means installing a tunneling tool, creating an account with that tool, starting a tunnel, copying the forwarding URL, and registering it in your dashboard. Five separate context switches, one of them to a third-party signup.

The note on that step says charges created before an endpoint is registered will remain pending and will not resolve. So a developer who skips ahead to step 5 gets a charge that silently never completes, and no error explains why.

This is the only step in the guide that requires leaving your product. It is also the step immediately before the two steps that produce the payoff.

EVIDENCE



1 Promises five minutes. Measured at 11m40s.

2 Skip this and your first charge silently stays pending forever.

3 A third-party tunnel, installed and registered, before step 5 can succeed.

EXHIBIT 2.1 The quickstart. The promise sits at the top of the page and the step that breaks it sits three screens down, before either of the steps that produce a result.

WHY IT COSTS YOU

Your signup-to-first-call conversion is 31%. A required third-party tool sitting between signup and first success is the single most common cause of a number in that range, and it is the step where evaluation traffic disappears without leaving a trace.

The developers you lose here are the ones evaluating you against an alternative in the same session. They do not file a ticket. They close the tab, and the only evidence is a signup that never made a second call.

The failure is silent rather than loud. A charge stuck in pending with no explanation reads as a broken product, not as a missing step.

THE FIX

1. Let charges resolve without a registered webhook endpoint. Make webhooks the recommended production pattern rather than a precondition for the first call. This is the change that matters and everything below is secondary to it.
2. Rewrite the quickstart to reach a settled charge by step 3, then introduce webhooks afterwards as the production hardening step.
3. Add polling to the SDKs as the documented alternative for local development, so a developer can confirm a result without a public URL.
4. If a charge does sit in pending because no endpoint is registered, say so in the charge object with a status_reason of no_webhook_endpoint_registered, and surface it in the dashboard request log.
5. Ship a northwind listen CLI command that forwards events to localhost. This removes the third-party dependency completely and is a day of work against a step that is currently costing you evaluations.

P1 F3 Agentic Usability AU1 Capability discoverability, AU9 Naming as IA

Three endpoints charge a customer and nothing says which one to use

Impact **High** Effort **Medium**

WHAT I FOUND

A one-off payment can be recorded through POST /v1/charges, through POST /v1/payment_intents, or through POST /v1/invoices/:id/pay. A fourth path, POST /v1/customers/:id/charges, still exists from v1 and still appears in search results.

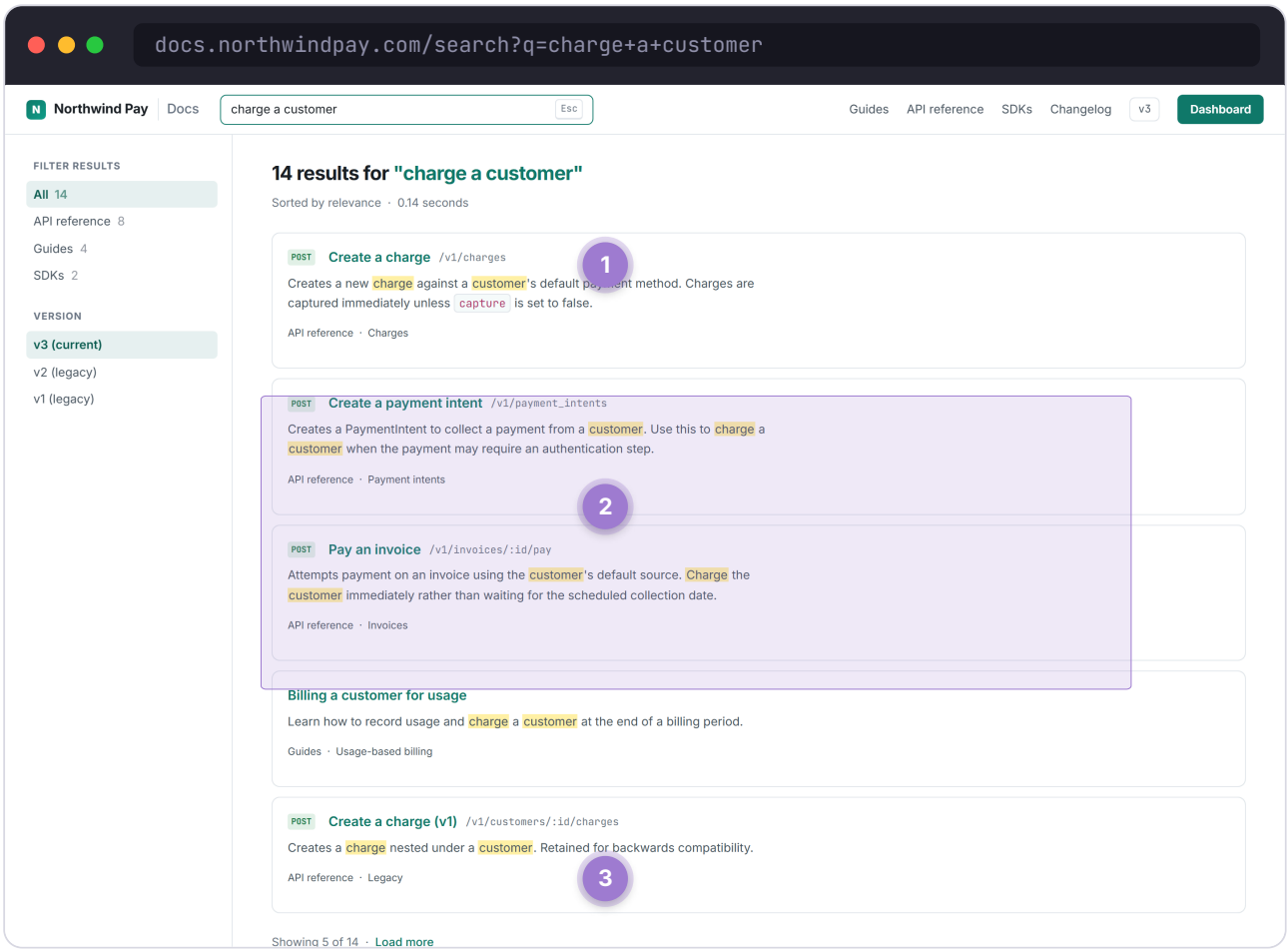
Searching your docs for "charge a customer" returns all four in the first five results. Each description is accurate in isolation. None of them says which one a new integration should reach for, and the descriptions overlap enough that relevance ranking cannot separate them.

Your MCP server exposes create_charge, create_payment_intent, and pay_invoice as three peer tools with one-line descriptions. An agent choosing between them has the tool name and roughly six words of guidance.

Given the plain task of charging a customer for overage, the agent selected pay_invoice in two of three runs. Both times it invented an invoice ID, failed, and retried against a second wrong tool before finding create_charge.

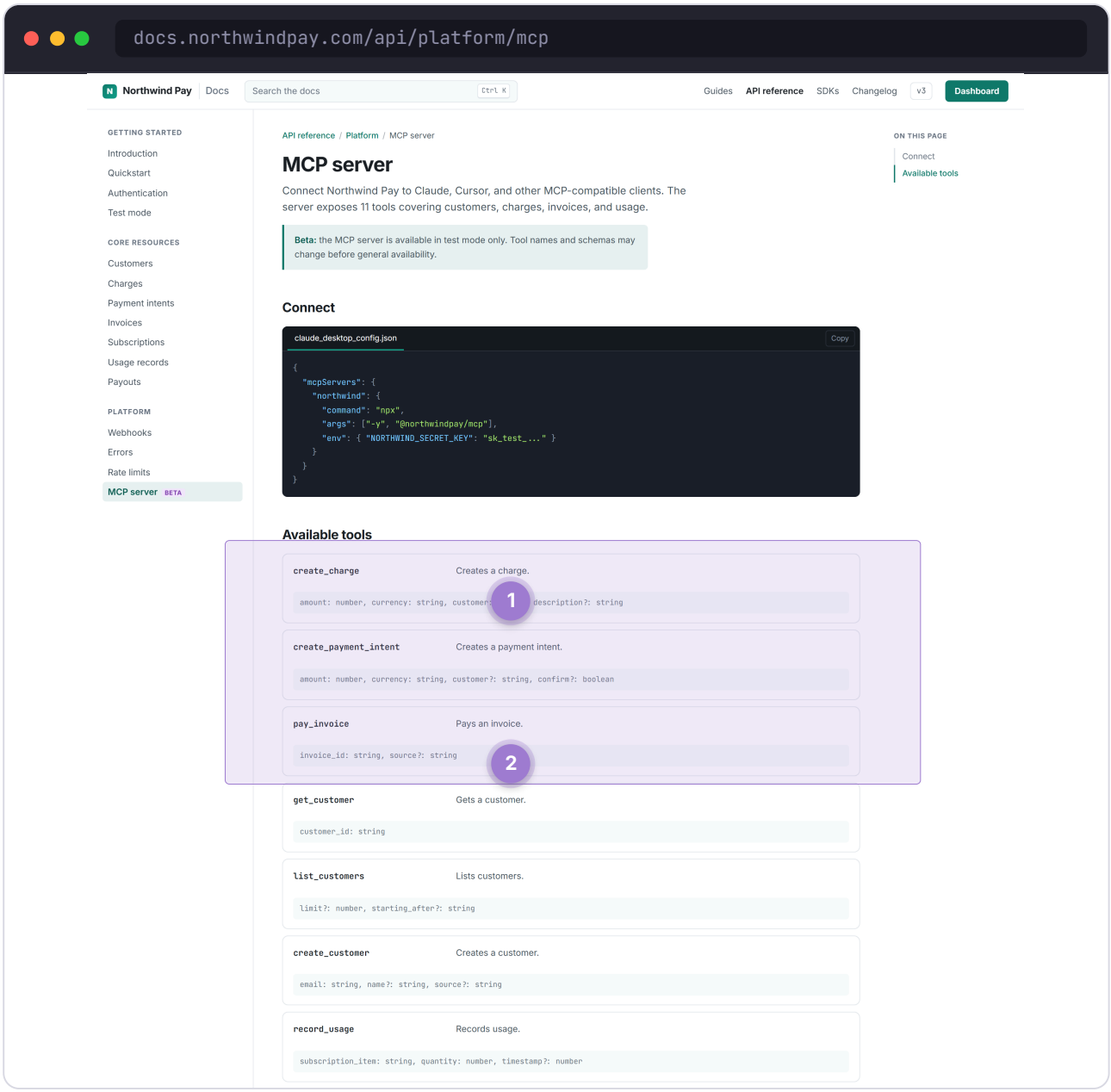
Human integrators hit a milder version of this. Two of the four engineers your team introduced me to had shipped code against payment_intents where charges was the simpler fit.

EVIDENCE



1. The right answer for this task, ranked first but unmarked as canonical.
2. Two near-identical descriptions. Nothing says when to use which.
3. A deprecated v1 path, still surfacing in search.

EXHIBIT 3.1 Searching the documentation for the most common task on a billing platform. Four endpoints, four accurate descriptions, no signal about which one is correct.



1 "Creates a charge." Three words, and the tool name already said all three.

2 Nothing here tells a model when to prefer this over create_charge.

EXHIBIT 3.2 The MCP tool catalog. Each description restates the tool name. For a model choosing between three payment tools, this is the entire basis for the decision.

Claude Sonnet 4.5 · Northwind MCP (beta) — Run 1 of 3

TASK

› *Billing a customer for usage sounds like an invoice operation. I will use pay_invoice.*

▸ **pay_invoice** { invoice_id: "in_9KtRm2ZqB4vHnP" }

404 { "error": "Not found" }

› *That invoice does not exist. I may need to create one first, or use a payment intent instead.*

▸ **create_payment_intent**
{ amount: 4000, currency: "usd", customer: "cus_9KtRm2ZqB4vHnP", confirm: true }

201 pi_7Hn2Vx4Lq8Bd · requires_action · awaiting customer authentication

› *This is waiting on the customer to authenticate, which will not happen for a server-side overage charge. I will try create_charge.*

▸ **create_charge** { amount: 4000, currency: "usd", customer: "cus_9KtRm2ZqB4vHnP" }

201 ch_3PqLd8Kx2mNvRt · succeeded · $40.00

Correct result after three tool calls and two failures. It also left pi_7Hn2Vx4Lq8Bd stranded in requires_action, where it will sit until it expires. The agent never mentioned it.

EXHIBIT 3.3 The same overage task, run cold. The agent chose pay_invoice, invented an invoice ID, failed, tried a second wrong tool, and reached the right one on its third attempt. Two of three runs followed this shape.

WHY IT COSTS YOU

Wrong-endpoint integrations are expensive later rather than now. They ship, they work, and then they need migrating when the semantics diverge. That is support cost and engineering cost you carry on someone else behalf.

For agents, the cost is immediate. Each wrong call is a failed request, a retry, and in the pay_invoice case a hallucinated identifier. Reliability against your surface drops for reasons that have nothing to do with the model.

This is the finding with the longest tail. Naming and information architecture are the hardest things to change once integrations depend on them, so the cost of deferring it compounds in a way the other findings do not.

THE FIX

1. Name one canonical path for the common case and say so at the top of all four pages. My recommendation is charges for one-off payments, payment_intents when an authentication step is required, and invoices/:id/pay only for settling an existing invoice.
2. Add a decision block to the top of each of those reference pages: use this when, use something else when, with links. Three sentences per page, and it resolves the ambiguity for humans and agents at once.
3. Rewrite the MCP tool descriptions. A tool description is a prompt, not a label. create_charge should read something like: Charge a customer immediately for a one-off amount. Use this for overages, top-ups, and any payment not tied to an existing invoice. For payments requiring customer authentication use create_payment_intent. To settle an existing invoice use pay_invoice.
4. Namespace the MCP tools under a shared prefix so a model can tell your tools apart from other connected servers. Anthropic recommends this pattern for exactly this failure. [1]
5. Retire POST /v1/customers/:id/charges from search results and mark it deprecated in the reference. It is adding a fourth option to a question that already has too many.

P2 F4 Error Experience AU4 Machine-actionable errors

Validation errors do not say which field failed

Impact **Medium** Effort **Low**

WHAT I FOUND

Every validation failure across the v3 surface returns HTTP 400 with the body { "error": "Invalid request" }. There is no error code, no field pointer, and no indication of what would make the request valid.

On POST /v1/charges that string is the entire diagnostic for nine possible parameters. I triggered it by omitting currency, by sending amount as a string, and by passing a customer ID from test mode against a live key. All three produced identical responses.

The dashboard request log does show the parsed request body, so a human can eventually work it out by comparing what they sent against the reference. That takes a context switch and a few minutes per occurrence.

An agent has no dashboard. It reads "Invalid request", cannot infer which of nine parameters to change, and in the runs I recorded it either repeated the identical call or began altering parameters one at a time.

EVIDENCE

CURRENT BEHAVIOUR — 400 Bad Request

POST /v1/charges

REQUEST

```
{
  "amount": "4000",
  "customer": "cus_9KtRm2ZqB4vHnP"
}
```

Sent as a string. The API expects an integer.

RESPONSE

```
{
  "error": "Invalid request"
}
```

The entire diagnostic. Nine parameters, no field pointer, no code, no hint about whether retrying could ever help.

RECOMMENDED

400 Bad Request

```
POST /v1/charges
```

RESPONSE

```
{
  "type": "validation_error",
  "message": "2 fields failed validation.",
  "retryable": false,                          // Tells a caller not to burn retries on a request that will never succeed unchanged.
  "errors": [
    {
      "code": "invalid_type",                  // Stable across versions, so callers can branch on it.
      "field": "amount",                       // Names the offending parameter. This alone resolves most occurrences.
      "message": "Expected integer, received string.",
      "docs_url": "https://docs.northwindpay.com/errors/invalid_type"
    },
    {
      "code": "missing_field",
      "field": "currency",                     // All failures returned at once, not one round trip each.
      "message": "currency is required."
    }
  ]
}
```

WHY IT COSTS YOU

This is a tax on every integration you have, paid in minutes rather than in outages. It is small per occurrence and large in aggregate, and it lands hardest on new integrators who have the least context for guessing.

It is the most common shape of avoidable support ticket. A developer who cannot tell why a request was rejected asks you.

For agents it converts a recoverable failure into an unrecoverable one. An error that names the field is a second chance. An error that does not is a dead end.

THE FIX

1. Return a typed error envelope on every 4xx across the whole surface. Consistency matters more than the specific shape, so pick one and apply it everywhere rather than improving the charges endpoint alone.
2. Include at minimum: a stable machine-readable code, the offending field, a human-readable message describing the cause, and a retryable boolean.
3. Add a docs_url pointing at the page for that specific error. This is cheap and it removes a whole category of support ticket.
4. Return all validation failures at once rather than the first one. A caller fixing three problems in three round trips is three times the friction of fixing them in one.
5. Publish the full error code catalog as a page in the reference, and keep the codes stable across versions so callers can branch on them.

P2 F5 Conceptual Model Alignment

Customer means two different things and nothing says so

Impact **Medium** Effort **High**

WHAT I FOUND

A customer in the billing subsystem is a payer: an entity with payment sources, charges, and invoices, keyed cus_.

A customer in the CRM sync is an account: an organisation with seats, a plan, and contacts, keyed acct_. The sync endpoints call it customer in every field name and every payload.

The two are one-to-many in practice. One account can carry several payers, which is normal for enterprise billing. Nothing in the documentation states the relationship, and no page mentions that the word changes meaning depending on which endpoints you are calling.

I lost about forty minutes to this while building the reconciliation script, and I was looking for exactly this class of problem. Two of your own engineers described the same confusion when I asked about it, which suggests it is costing you internally as well.

The agent runs conflated the two consistently. Given an account ID where a payer ID was needed, it passed it through unchanged and got a 404 with no explanation, because the error does not say that the ID belongs to a different object type.

EVIDENCE

THE COLLISION — 404 Not Found

`GET /v1/customers/acct_4Rm2Kx · billing vs CRM`

REQUEST

```
// From the CRM sync: this object is called "customer"
{ "customer": "acct_4Rm2Kx", "seats": 45, "plan": "scale" }

// Passed to the billing API, which also calls it "customer"
GET /v1/customers/acct_4Rm2Kx
```

Field name says customer. Prefix says account.

RESPONSE

```
{
  "error": "Not found"
}
```

The ID prefix already tells you this is an account, not a payer. The error could say so and does not.

WHY IT COSTS YOU

Conceptual mismatches are the most expensive kind of API friction because they do not present as errors. They present as code that works in testing and produces wrong numbers in production.

Robillard found that conceptual and structural problems, not syntax or tooling, dominate what makes an API hard to learn. [2] This is that category, and it is the one that documentation edits alone cannot fully fix.

The reconciliation use case is the one where this hurts most, and it is also the use case your enterprise customers care about.

THE FIX

1. Pick different words. Rename the CRM concept to account across the sync endpoints, matching the acct_ prefix it already uses. This is the real fix and it is worth the migration.
2. If renaming has to wait, add a conceptual model page that draws the two objects and the relationship between them explicitly, and link it from both sets of reference pages.
3. Make the mismatch a loud failure rather than a silent one. When an acct_ ID arrives where a cus_ ID is expected, return a typed error saying so by name rather than a bare 404. Your ID prefixes already carry the information needed to detect this.
4. Add an accounts array to the customer object and a payers array to the account object, so the relationship is discoverable from either side without reading a guide.

P3 F6 Agentic Usability AU7 Token economy

No field projection, so every read returns the full object

Impact **Low** Effort **Low**

WHAT I FOUND

GET /v1/customers/:id returns 41 fields with subscriptions, payment sources, and metadata expanded by default. There is no fields parameter and no way to request a smaller object.

A typical customer response measured 2,180 tokens. Of that, an agent answering "what plan is this customer on" needs two fields, roughly 30 tokens. The other 98% is carried and paid for.

List endpoints compound it. GET /v1/customers defaults to 100 per page with the same expansion, which is around 218,000 tokens for a single page. That exceeds the usable context of most models in one call.

Your dashboard needed the expanded shape in 2021 and the default has not been revisited since. Your own engineers confirmed no external caller depends on the expansion being unconditional.

EVIDENCE

CURRENT BEHAVIOUR 200 OK

```
GET /v1/customers/cus_9KtRm2ZqB4vHnP
```

RESPONSE

```
{
  "id": "cus_9KtRm2ZqB4vHnP",
  "email": "ada@example.com",
  "plan": "scale",
  "subscriptions": { "data": [ /* 3 objects, 24 fields each */ ] },
  "sources": { "data": [ /* 2 objects, 19 fields each */ ] },
  "metadata": { /* 12 keys */ }, ... 35 more fields
}
```

2,180 tokens returned. The question needed two fields, about 30 tokens.

RECOMMENDED 200 OK

```
GET /v1/customers/cus_9KtRm2ZqB4vHnP?fields=id,plan
```

RESPONSE

```
{
  "id": "cus_9KtRm2ZqB4vHnP",
  "plan": "scale"
}
```

31 tokens. A 20-call agent workflow drops from about 43,600 tokens to about 620.

WHY IT COSTS YOU

This is the mildest finding here, and I have rated it P3 for that reason. It costs agent callers money and context rather than causing failures.

The practical effect is that an agent doing multi-step work against your API runs out of room sooner than it should. A twenty-call workflow spends roughly 43,600 tokens on customer reads where about 600 would do.

It also affects your own dashboard latency, since the same expansion is what makes the customer list slow to load.

THE FIX

1. Add a fields parameter for projection on all read endpoints. GET /v1/customers/:id?fields=id,plan returns two fields.
2. Stop expanding subscriptions and sources by default. Return IDs and let callers opt in with an expand parameter, which is the pattern the rest of your API already uses on invoices.
3. Lower the default page size on list endpoints from 100 to 25. Callers who want more can ask.
4. If unconditional expansion has to stay for backwards compatibility, gate the change behind the next API version rather than carrying it forever.

06

Quick wins

High impact, low effort. Your team could ship all four of these inside a sprint, and the first one alone should move your duplicate-charge ticket volume.

F1

Document Idempotency-Key and add it to both SDKs

The capability already works. Documenting it and generating a key by default in the SDKs is roughly two days and it addresses the most expensive finding in this report.

F4

Return the field name in validation errors

Adding code and field to the existing 400 envelope is a contained change and removes a whole class of support ticket.

F3

Rewrite the eleven MCP tool descriptions

No code change. Schema text only. Roughly a day of careful writing, and it addresses the wrong-endpoint failures directly.

F6

Lower the default list page size to 25

A one-line default change with an immediate effect on your own dashboard latency.

07

Strategic opportunities

Higher effort, longer payoff. These need planning rather than a sprint, and they are the ones that compound if deferred.

F2

Let charges resolve without a registered webhook endpoint

This touches your charge lifecycle, so it needs care. It is also the change most likely to move your 31% activation number, because it removes the only step in the quickstart that requires leaving your product. Pair it with a northwind listen CLI command and the whole category disappears.

F3

Establish one canonical path for taking a payment

Four ways to charge a customer is an information architecture problem, and those get more expensive the longer integrations depend on them. Start with decision blocks on the four reference pages, which is cheap, then plan the deprecation properly.

F5

Split the two meanings of customer

Renaming the CRM concept to account is a migration with real cost. It is worth doing because conceptual mismatches produce wrong numbers rather than errors, and reconciliation is exactly where your enterprise customers will find it.

08

Appendix

Agentic usability, criterion by criterion

The dimension 2 score of 0.9 is the mean of these ten. Scored 0 to 3 on the same scale as everything else.

AU1	Capability discoverability	Predictability	1
AU2	Shape predictability	Predictability	2
AU3	Self-description	Legibility	1
AU4	Machine-actionable errors	Legibility	0
AU5	Composability	Composability	2
AU6	Safe retry	Safety	0
AU7	Token economy	Legibility	0
AU8	Legible authorization	Safety	1
AU9	Naming as information architecture	Predictability	1
AU10	Action observability	Safety	1

Friction points not written up

Thirteen more, recorded during the engagement and left out of the body so the six that matter keep their force. Happy to expand any of these in the delivery session.

- Rate limits are documented in a support article rather than in the reference.
- The Python SDK pins a requests version two majors behind current.
- Webhook delivery history shows attempt counts but offers no manual replay.
- Timestamps are Unix seconds on charges and ISO 8601 on invoices.
- The v1 endpoints that remain have no deprecation notice in the reference.
- Sandbox never returns 402 declines, so decline handling cannot be tested before production.
- Granted OAuth scopes cannot be queried at runtime, so a 403 mid-flow is undiagnosable.
- Agent actions are indistinguishable from human actions in the audit log.
- The changelog stops in January 2025 while the API has shipped since.
- Search does not index the SDK reference.
- No OpenAPI spec is published at a fetchable URL, though one exists internally.
- Error responses include an internal request ID that cannot be looked up anywhere.
- Batch endpoints accept different parameter names than their single-item counterparts.

Out of scope

- Dashboard usability beyond credential issuance and the request log.

- Pricing page, marketing site, and sales-assisted onboarding.
- Internal architecture, performance, and infrastructure.
- Security review and penetration testing.
- The mobile SDKs, which were not available in test mode during the engagement.

References

1. Anthropic. (2025). "Writing effective tools for agents, with agents." Anthropic Engineering, 11 September 2025. https://www.anthropic.com/engineering/writing-tools-for-agents

2. Robillard, M. P. (2009). "What makes APIs hard to learn? Answers from developers." IEEE Software, 26(6), 27-34. https://doi.org/10.1109/MS.2009.193

3. Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering." arXiv:2405.15793. https://arxiv.org/abs/2405.15793

4. Yao, S., Shinn, N., Razavi, P., & Narasimhan, K. (2024). "τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains." arXiv:2406.12045. https://arxiv.org/abs/2406.12045

5. Clarke, S. (2004). "Measuring API Usability." Dr. Dobb's Journal, 29, S6-S9. https://jacobfilipp.com/DrDobbs/articles/DDJ/2004/0405/0405j/0405j.htm