Use agents to expose interface ambiguity

Key takeaways

1. Give the agent the interface, callers, tests, and product intent before coding.
2. Let the agent surface choices; humans approve caller-visible policy.
3. Require executable evidence and a report of failures or contradictions.

Common misunderstandings of specs

Which claim would you defend right now? Choose before the evidence.

"An algorithm name plus a type signature is enough to define a module."

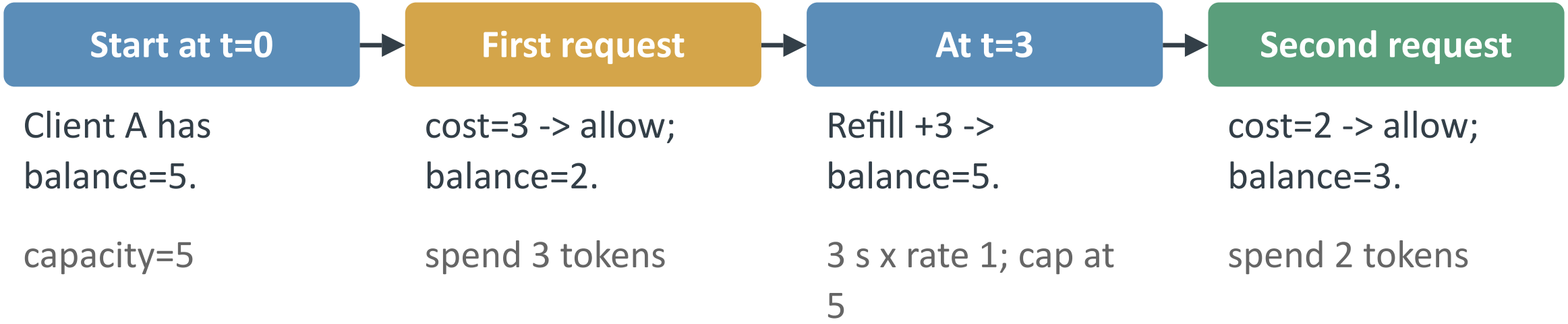
"The human must find every ambiguity and verify every invariant in the code."

"A detailed file-by-file task list is a complete implementation plan."

Keep your choice. Look for the mechanism or counterexample that would make it fail.

A token bucket spends and refills a balance

Each client has its own balance; an affordable request spends cost, while a denial spends nothing



now=3 is the current call’s timestamp; cost=2 is how many tokens that call must spend.

source: spec-author F1 calibration: specs/medium.md

Typed specifications can still be ambiguous

Broad types do not define valid inputs, errors, output guarantees, or state effects

```
Implement a token-bucket rate limiter.

Each key has an independent bucket that starts full. Before a
request, elapsed time refills it at rate, capped at capacity.
If tokens >= cost, allow and subtract cost; otherwise deny
without spending tokens.

allow(key: Key, now: Seconds, cost: Tokens) -> Decision
Decision: allowed, retry_after, remaining.
```

- Can Tokens be zero or negative?
- Is cost > capacity invalid or permanently denied?
- What do denial and equal-time calls guarantee about state and ordering?

Specify valid inputs, result variants, state changes, time behavior, and ordering.

source: spec-author F1 calibration: specs/medium.md

One request admits two reasonable builds

Both agents can explain their choice; neither can recover policy that was never written

Agent A: invalid requests are errors

```
if cost <= 0:
    raise ValueError("cost")

if cost > capacity:
    raise ValueError("cost")
```

Agent B: all costs produce decisions

```
if cost <= 0:
    return Decision(True, 0, tokens)

if cost > capacity:
    return Decision(False, inf, tokens)
```

Reasonableness cannot choose which behavior your callers should depend on.

Six agents built the same vague contract

Two Haiku, two Sonnet, one Opus, and one GPT-5.5 each implemented the prose and ran the same 22-event trace

Unspecified input	What the six builds did	Required semantics	Risk to callers
cost = 0	6/6 admitted a no-op	reject before state access	invalid input looks successful
cost = −1	5/6 admitted it; 4 raised balance above capacity	reject before state access	capacity invariant breaks
cost = 10, capacity = 5	5/6 returned a finite retry time	deny as never retryable	caller schedules a retry that cannot succeed

Shared conventions made the builds agree; agreement did not define correctness.

source: spec-author F1 calibration: README.md and results/dump_vague.txt

Unchosen semantics spread to callers

Once consuming software trusts the result, changing the semantics later is a breaking API change

Open contract	Agent default	Consuming software
cost > capacity has no defined outcome.	Return Denied(retry_after=5).	Schedule a retry at t+5; the request can never fit.
No semantic commitment	A guess becomes API behavior	Bad behavior moves downstream

A missing commitment at the boundary becomes behavior every caller must now handle.

Brief the agent, then ask for an audit

The agent expands the case space; the human still approves product semantics

Boundary-audit prompt

```
Read rate_limiter.py, scheduler.py, and
tests/test_rate_limiter.py. Do not edit yet.

Audit RateLimiter.allow as a public boundary. Return a numbered
table with one row per normal, boundary, or invalid input class:

- candidate output or exception;
- state before and after, plus the invariant;
- time, ordering, retry, and caller risks;
- a stricter domain type, when useful.

Do not choose product policy. Mark each choice for my approval.
```

Have the agent expose cases and choices; reserve product policy for human approval.

Approve only the caller-visible choices

The agent finds the cases and consequences; the human chooses product policy

Rule	Case the agent found	Caller-visible choice	Approved policy
R1	cost <= 0	error | Invalid result	Reject before calling the limiter
R2	cost > capacity	error | finite retry | never retry	Denied(NeverRetry); scheduler must not queue
R3	two calls at the same now	call order | unspecified	Evaluate in call order

The human chooses three policies; the agent records and traces them through the project.

Types encode the approved boundary

R1 excludes invalid costs; R2 gives callers an explicit never-retry result

Input boundary - R1

```
@dataclass(frozen=True)
class PositiveTokens:
    value: float

    def __post_init__(self):
        if self.value <= 0:
            raise ValueError("positive cost
required")

@dataclass(frozen=True)
class Request:
    key: ClientKey
    now: MonotonicTime
    cost: PositiveTokens
```

Output boundary - R2

```
@dataclass(frozen=True)
class Allowed:
    remaining: Tokens

@dataclass(frozen=True)
class NeverRetry:
    pass

@dataclass(frozen=True)
class Denied:
    remaining: Tokens
    retry_after: SecondsUntilRetry |
NeverRetry

Decision = Allowed | Denied
```

Test the contract through a real caller

R2 requires two observable facts: the limiter returns NeverRetry and the scheduler enqueues nothing

```
def test_oversized_request_is_not_rescheduled() -> None:
    limiter = RateLimiter(rate=1, capacity=5)
    scheduler = Scheduler()
    request = Request(
        key=ClientKey("A"),
        now=MonotonicTime(0),
        cost=PositiveTokens(6),
    )

    result = limiter.allow(request)
    scheduler.handle(result)

    assert result == Denied(
        remaining=Tokens(5), retry_after=NeverRetry()
    )
    assert scheduler.pending == []
```

Humans choose policy; agents trace and verify

The human reviews choices and exceptions, not every invariant or changed line

Your Responsibility	Agent's Responsibility
Choose among the caller-visible policies the agent surfaces.	Read the interface, callers, and existing tests.
Confirm the selected behavior matches product intent.	Enumerate cases and trace consequences across the project.
Decide whether unresolved risks block the change.	Implement approved rules, run checks, and report contradictions.

Human review targets: open choices, failing checks, and unresolved contradictions.

Include the rule table in the coding prompt

The agent maps them to code, tests, and callers; the human does not hold them in memory

Activity-only prompt

```
Implement RateLimiter.allow.
Update the tests and scheduler.
Run pytest.
```

Rule-linked prompt

```
Implement approved rules R1-R3.
Read scheduler.py before editing.

Add one test per rule and one
limiter-to-scheduler scenario.

Return: rule -> test -> result ->
caller effect. List contradictions
and unresolved risks.
```

Example: review the agent’s exception report

Runnable tests and caller locations direct human attention to one unresolved policy conflict

Check	Agent-provided evidence	Result	Needs human attention?
R1 invalid cost	test_positive_tokens.py	PASS	No
R2 oversized request	test_scheduler.py::test_never_retry	PASS	No
R3 same-time order	test_rate_limiter.py::test_call_order	PASS	No
Caller scan	batcher.py assumes a finite retry	CONFLICT	Choose compatibility or change caller

Change what the human reviews

1. Ask the agent to discover ambiguous cases from the interface, callers, and tests; do not brainstorm them alone.
2. Approve only caller-visible product policy, and store those choices in a durable rule table.
3. Require the agent to encode, test, trace, and report exceptions; inspect code where risk warrants it.