How assumptions become architecture

Key takeaways

1. An incomplete request can leave a design decision to the agent. A plausible answer may commit the rest of the system to that choice.

2. Callers and tests accumulate around the first implementation. Changing a small decision then requires changing the code that depends on it.

3. State the chosen policy, ask for evidence through callers, and resolve conflicting assumptions the agent finds elsewhere in the project.

Common misunderstandings of interface decisions

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

"Naming an algorithm settles the behavior of its corner cases."

"Passing a module's tests establishes that its callers follow the intended policy."

"A small decision in one function will be cheap to reverse later."

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

Token buckets limit rate while allowing bursts

A token bucket is a rate limiter. Each client holds tokens, which are credits that requests spend to use a service.

Capacity 5; refill 1 token/s; starts full.

Time	Cost	Decision	Tokens
0 s	3	Allow	5 → 2
0 s	3	Deny	2 → 2
1 s	3	Allow	3 → 0
10 s	—	Idle	5 (cap)

HOW IT WORKS

Time adds tokens up to capacity. Idle time cannot accumulate credit beyond that cap.

If the balance covers the cost, allow the request and subtract its cost. Otherwise deny it without spending tokens.

Capacity bounds a burst. The refill rate bounds sustained use.

The request leaves one decision open

The interface

```
allow(key, now, cost) -> Decision
```

Every returned Decision has all three fields:

Field	Meaning
allowed	True if admitted; False if denied
retry_after	Seconds until this request can be admitted; 0 if allowed
remaining	Tokens left after this call

WHAT IS STILL MISSING?

Is a cost larger than capacity an invalid input, or a normal denial?

If it is denied, which retry_after value means this request can never be admitted?

What should the caller receive when a request costs 10 tokens?

Instapoll

Instapoll · Review · Sep 22 A

An agent proposes returning a finite retry delay for every denied request. You plan to build a scheduler that automatically retries each denial. Which unanswered question could invalidate this retry strategy?

- A. Should simultaneous retries run in arrival order or priority order?
- B. Can every denied request eventually succeed through waiting alone?
- C. Should the bucket refill on a timer or when a request arrives?
- D. Should queued requests survive a restart of the scheduler process?

A permanent denial gets a retry time

We asked six agents to implement the same token-bucket interface.

1. **We submit the same request to every implementation**
   Capacity 5, refill 1 token/s, initially full. The request costs 10 tokens.

2. **Five implementations tell the caller to retry in 5 seconds**
   They return allowed = False and retry_after = 5: deny now, try again later.

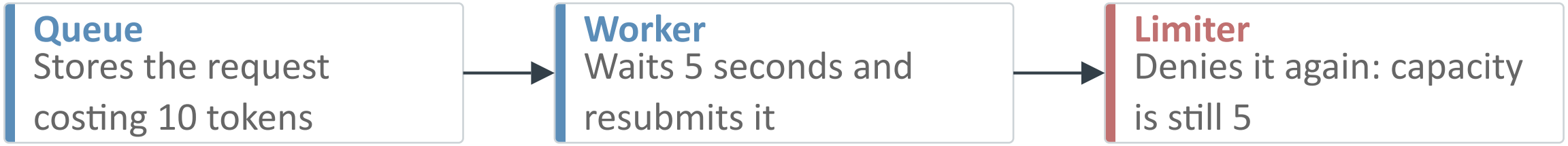
3. **Waiting cannot make this request succeed**
   After 5 seconds, the balance is still at most 5. The request still costs 10.

The missing policy became an API choice: treat an oversized request as retryable.

The agent builds the wrong retry path

Illustrative continuation: the caller is built around that retry result

Agent: “Each denial tells me when to retry. I’ll add a queue to hold denied requests and a worker to resubmit them when the delay expires.”



The agent extends the retry assumption instead of revisiting it. Correcting the policy now also requires changing which requests the caller queues.

The API guess now supports a retry path that oversized requests should never enter.

Follow the request past the passing test

Illustrative continuation: diagnosing the retry queue

Capacity 5, refill 1 token/s, initially full. The test submits a request costing 10.

Agent “The test passes. The request was rejected.”

Student “What happens after that? Show me the queued request’s next attempt.”

Attempt	Token balance	What the caller does
Now	5	Gets a denial; queues a retry in 5 s
5 seconds later	5	Gets a denial; queues another retry

The test checked rejection. The trace reveals a retry that makes no progress.

Would waiting longer help?

Student	“Maybe five seconds is too short. Would waiting a minute help?”
Agent	“No. The balance stops at 5 tokens, however long we wait. This request needs 10. But with a balance of 2, a request costing 3 could succeed after a refill.”
Student	“Then stop retrying requests that exceed capacity. Keep retries for requests that only need more tokens.”

The trace suggests a problem; the capacity limit explains why it persists.

Make the API distinguish the two denials

Student	“How will the caller know whether waiting can help?”
Agent	“Change retry_after to allow None, meaning do not retry. A delay in seconds still means try again later.”
Student	“Use that. Add these two tests, including what gets queued.”

New expected results: capacity 5, refill 1 token/s. Both requests are denied.

Test input	retry_after	Expected caller action
Cost 3; balance 2	1 second	Queue a retry
Cost 10; balance 5	None	Reject; queue nothing

The API gains a permanent-denial result. The tests also protect ordinary retries.

Another caller still assumes retries

Illustrative continuation: applying the changed API across callers

Agent "The scheduler passes both new tests. The batch submitter fails: it queues every denial, and its old test expects that."

Student "Make it follow the same rule. Replace that old expectation, then run both cases through both callers. Show me what each does."

Agent's results after updating the batch submitter

Caller	Cost 3; balance 2	Cost 10; balance 5
Scheduler	Queues a retry in 1 s	Rejects; queues nothing
Batch submitter	Queues a retry in 1 s	Rejects; queues nothing

Changing the result also requires changing the callers and tests built around it.

The commitment grows with its callers

1. One plausible answer can silently settle an unresolved requirement. Here, a finite retry delay led the caller to queue a request whose token cost exceeded the bucket’s capacity.

2. The commitment grows as other code depends on it. A different answer then changes the scheduler and its tests, even though the original decision occupied one line.

3. State the rule in terms of caller behavior. Ask the agent to demonstrate both denial paths through callers and surface conflicting assumptions. Resolve those conflicts before treating the change as complete.

Is this a configuration mistake?

10 min

- Choose two cases. Each arrow shows the agent's proposed new code.
  - 1. API calls time out → a background retry worker
  - 2. Database writes get read-only errors → a queue to hold writes for later
  - 3. Expected records are missing → a service to copy records between databases
  - 4. Large uploads are rejected → code to split and reassemble uploads
  - 5. Data disappears after each restart → an export-and-restore mechanism
  - 6. Displayed times are off by six hours → code to rewrite stored timestamps
  - 7. The browser blocks API calls → a new proxy service
  - 8. Unchanged files rebuild every time → a custom build cache
- In pairs, propose a configuration mistake for each case. Write a message asking the agent to test it and return evidence.
- Compare with another pair. Which proposed code becomes unnecessary if your explanation is confirmed?