COPILOT

A Smart Solana Transaction Infrastructure

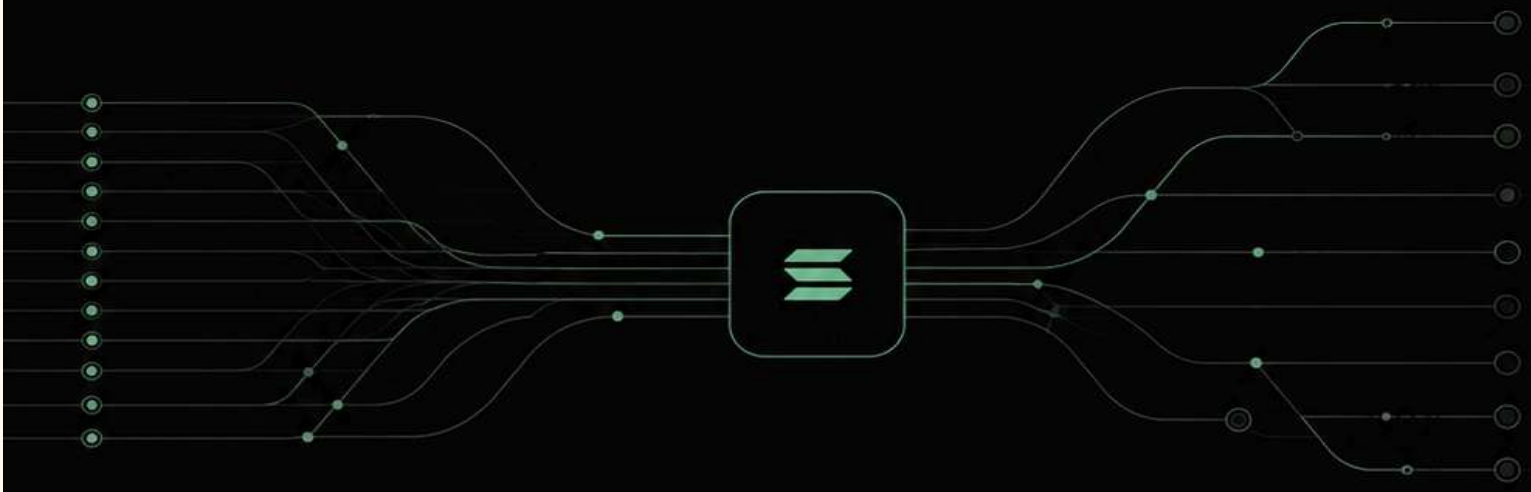


Table of Contents

Introduction

Copilot is a Rust workspace that takes a bundle from tip pricing through onchain finality and recovers from failure through autonomous AI-driven retry. It is built around the Jito Block Engine and the Yellowstone gRPC streaming protocol. This document is an engineering record grounded in actual source code and the lifecycle logs from 56 mainnet runs.

The Solana fee market is an auction. Jito's bundle ordering layer runs a second, higher-stakes auction on top of it. The conventional approach to transaction sending uses a static tip, a stale blockhash, and polling for confirmation. This fails in four specific ways: stale tip pricing that misses auction shifts, stale blockhashes that push transactions near or past their 150-slot validity window, silent failure from the Block Engine which returns HTTP 200 even for bundles it will never forward, and uninformed retry that conflates causes requiring different responses.

Copilot's answer is to move every decision onto live data: tips from the actual Jito landed-tip distribution sampled at request time, blockhashes from the most recently processed block, landing confirmation from a persistent Yellowstone stream rather than a polling loop, and failure classification from a typed taxonomy applied to observable signals. The autonomous retry agent then uses Claude Sonnet to make the retry decision from that same live data rather than from a hardcoded policy.

The operational record confirms the thesis. Once `COPILOT_JITO_UUID` was configured (runs 16 onward), thirteen of forty-one subsequent bundles landed with submit-to-processed latencies between 318 and 1,683 milliseconds. The fault injection demo was run four times (inject sessions: runs 26/27, 51/52, 53/54, and 55/56). In every session the classifier correctly identified expired blockhash at 90 percent confidence. The agent correctly matched the fix to the cause in each case, adjusting the tip based on the live oracle snapshot rather than a hardcoded rule. Two of the four retry attempts landed (runs 27 and 56); two did not land within the deadline.

Notes:

- The range floor is run-56 (318 ms).
- The range ceiling is run-48 (1,683 ms), an outlier caused by a brief delay between Geyser subscription establishment and the first slot event; all other new landings are within 385-583 ms.
- Count breakdown: 41 runs = runs 16-56. 13 landings = runs 16, 17, 21, 23, 27, 28, 29, 35, 37, 41, 48, 49, 56. Four of the 41 are deterministic injected faults (runs 26, 51, 53, 55) that cannot land; two are agent retries that did not land (runs 52, 54).

1. System Architecture

1.1 The Eight-Crate Workspace

Copilot is a Cargo workspace with eight member crates. Each crate owns exactly one concern. There are no circular dependencies. The cli crate assembles all others into a binary.

```
copilot/
├── crates/
│   ├── geyser/      Yellowstone gRPC subscriber - maintains ChainState
│   ├── leader/      Epoch leader schedule - Jito-connected leader detection
│   ├── tip-oracle/  Dynamic Jito tip pricing - congestion classification
│   ├── bundle/      Bundle construction, submission, and status queries
│   ├── lifecycle/   Signature tracking - processed/confirmed/finalized timestamps
│   ├── fault/       Failure classification - deterministic fault injection
│   ├── agent/       Autonomous retry agent - Anthropic API client, reasoning log
│   └── cli/         Binary entry point; Stack assembly - all subcommands
├── rust-toolchain.toml   Pinned toolchain: 1.96.0 / edition 2024
└── .cargo/config.toml    macOS embed-bitcode=no (resolves LLVM 22 linker issue)
```

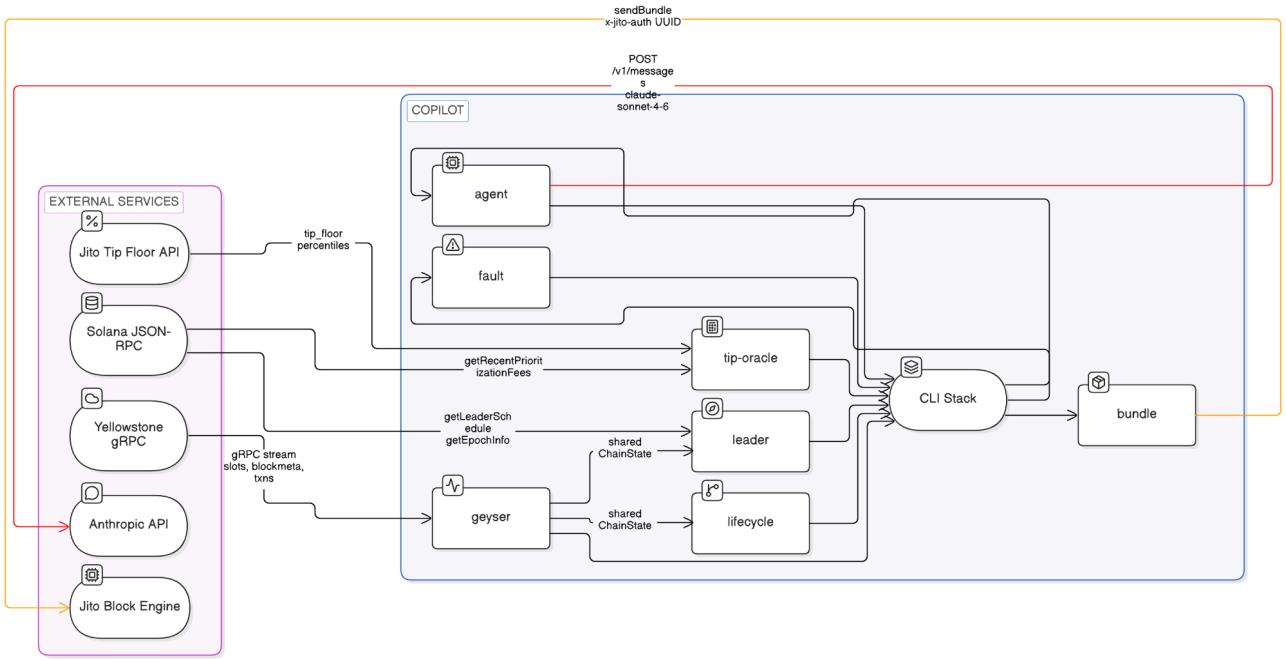
Crate	Responsibility	Key external dependencies
geyser	Yellowstone gRPC stream; lock-free ChainState	yellowstone-grpc-client, arc-swap
leader	Epoch leader schedule; Jito-connected slot detection	solana-rpc-client
tip-oracle	Jito tip floor percentiles; priority fee sampling; congestion classification	reqwest, solana-rpc-client
bundle	Versioned transaction construction; sendBundle submission; status queries	jito-sdk-rust, bincode, base64
lifecycle	Commitment-level tracking via geyser stream; log file writer	yellowstone-grpc-client
fault	Deterministic failure classification over typed signals; no async I/O	serde

agent	Anthropic Messages API client; decision parser; JSONL reasoning log	reqwest, serde_json
cli	Stack assembly; subcommand parsing; inject_demo orchestration	All seven above, clap, dotenvy

Only geyser (stream), leader (RPC), tip-oracle (RPC + HTTP), bundle (Block Engine), and agent (Anthropic API) make network calls. The fault crate has no async code by design: classification is a pure synchronous function, making it testable without a runtime.

1.2 System Architecture Diagram

The diagram below shows Copilot as a running process. Five external services feed into eight internal crates. ChainState (owned by geyser, shared via Arc) is the data hub: it holds the values that leader, lifecycle, and cli/Stack need from the gRPC stream without subscribing to it directly. The submission path runs through tip-oracle and bundle. The recovery path runs through fault and agent.



1.3 The Streaming Layer: Live Chain State

Why streaming over polling: The alternative to a persistent gRPC stream is polling getLatestBlockhash, getSlot, and getSignatureStatuses on a timer. That approach works for simple applications but fails on three dimensions critical to Copilot.

Dimension	Yellowstone gRPC	Solana JSON-RPC polling
Slot event latency	Events pushed within ~50 ms of commitment transition	Latency equals poll interval plus processing; typically 200 ms to 2 s
Blockhash freshness	BlockMeta event carries the hash and its originating slot	getLatestBlockhash at processed is current but stateless; at finalized it is 31 slots old
Signature confirmation	Filter updated in-stream; server delivers the event when the signature appears	Must be polled repeatedly; minimum latency is one poll cycle after landing
Signature subscription	Stream filter updated via a new SubscribeRequest on the open sink	Not possible; no equivalent stateless RPC method
Commitment visibility	Single stream delivers processed, confirmed, and finalized events independently	Separate calls required per commitment level

The signature subscription row is the structural constraint. The lifecycle crate detects landings by installing a filter on the open stream and waiting for the stream to deliver a transaction event for the watched signature. This mechanism does not exist in the JSON-RPC API. There is no way to replicate it with polling without knowing the target slot in advance.

Lock-free ChainState: The geyser crate writes chain data into Arc<ChainState>, which is shared by reference with every other crate that needs it. All fields are designed for concurrent read access without locking: three AtomicU64 slot counters (processed, confirmed, finalized), one ArcSwapOption<BlockhashInfo> (latest blockhash plus its originating slot), one watch::Sender<u64> (slot tip notifications), and one broadcast::Sender<u64> (landing notifications). Readers never block writers.

Reconnection and backpressure: The Yellowstone library's built-in AutoReconnect wrapper handles gRPC-level errors but silently stops reconnecting on a clean server-initiated close (it sets stop = true on Poll::Ready(None)). Copilot wraps this in an outer run() loop that re-enters the connection on any termination. Exponential backoff starts at 500 ms, doubles per attempt, caps at 30 seconds, and resets to 500 ms after any session in which at least one valid message was received. The subscription filter is re-established from the current ChainState values immediately after reconnect, so no state is lost. The broadcast::Sender for landing events has capacity 16; slow consumers get a Lagged error and skip to the newest value, which causes a missed landing event to be treated as a non-landing rather than a crash.

1.4 Leader Windows and Jito Submission Timing

Submitting a bundle to the Block Engine during a slot whose assigned validator is not Jito-connected means the bundle is not forwarded to that validator. To find the next Jito-connected slot, Copilot calls getConnectedLeaders on the Block Engine to get the current set of Jito-connected validator pubkeys, then scans the epoch leader schedule for the first matching slot within the next 256 slots (LEADER_LOOKAHEAD_SLOTS).

The epoch schedule is stored as a Vec<u16> index into a Vec<Pubkey> rather than a Vec<Pubkey> per slot. This reduces memory from about 13.5 MB to under 1 MB per epoch. Slot lookups are O(1). When getConnectedLeaders fails, it returns an empty set rather than an error; the pipeline logs the condition and submits anyway, relying on the Block Engine's internal routing.

Three outcomes are possible before each submission: a specific Jito leader slot is found and logged; the Jito set is unavailable (RPC failure); or no Jito leader appears within the 256-slot lookahead. In all three cases the bundle is submitted. The Block Engine can hold and forward bundles when a connected validator later becomes leader. Not submitting because no Jito leader is immediately visible would sacrifice potential landings. The leader window check is informational and diagnostic, not a gate.

2. The Submission Pipeline

2.1 Dynamic Tip Pricing

A hardcoded tip fails in both directions. When the network is quiet, it overpays: the p50 floor can sit below 1,000 lamports and a 10,000-lamport bid wastes 9,000 per submission. When the network is active, it underpays: congestion spikes can shift the median by an order of magnitude in seconds, and a bundle below the floor is silently dropped.

The TipOracle::snapshot() method samples both sources in parallel using tokio::try_join!: the Jito tip floor API (percentiles of recently landed bundle tips in SOL, converted to lamports) and getRecentPrioritizationFees (up to 150 recent slots of base-layer compute-unit prices).

Tip floor values from the fault injection session (slot 427,360,290):

Field	Value	Meaning
p25	1,000 lamports	25th percentile of recent landed bundle tips
p50	1,039 lamports	Median; the central auction reference
p75	5,000 lamports	Baseline Copilot uses for submissions
p95	87,378 lamports	Primary signal for congestion classification
p99	2,320,000 lamports	Extreme outliers; typically targeted MEV
ema_p50	1,771 lamports	Jito's pre-computed EMA of p50; used for median_rising

Congestion classification uses the tail ratio p95 / max(p50, 1) (not p99, which is too volatile for classification). The median_rising flag fires when p50 > ema_p50.

Level	Condition	Meaning
Low	ratio < 4 and median not rising	Calm auction; p75 is a safe bid
Moderate	ratio >= 4 and not rising, or ratio < 4 and rising	Some dispersion or upward pressure
High	ratio >= 20, or ratio >= 4 and rising	Significant tail activity or heating median
Severe	ratio >= 100	Extreme competition; MEV bots active for a high-value target

At the inject session: ratio = 87,378 / 1,039 = 84.1 (above the High threshold of 20), median_rising = false (1,039 < 1,771). Level: High.

Baseline tip is jito_tip_floor.p75, floored at MIN_TIP_LAMPORTS = 1,000. P75 places the bundle in the upper quartile of the distribution: it beats 75 percent of recent submissions without overpaying in calm markets. The user can override with copilot run --tip N.

2.2 Bundle Construction

A Jito bundle is one to five Solana transactions submitted as an atomic unit: all are included in a block or none are. Copilot's tip-only bundle uses a single transaction with three instructions in a fixed order.

#	Instruction	Program	Parameters
1	SetComputeUnitLimit	Compute Budget	units: 10_000 (TIP_ONLY_CU_LIMIT)
2	SetComputeUnitPrice	Compute Budget	micro_lamports: prio_fees.p50
3	SystemTransfer (tip)	System	from: payer, to: tip_account, lamports: tip_lamports

The tip transfer must be last. The Jito validator runs bundle instructions in order and rolls back the entire bundle if any instruction fails. Placing the tip last means the validator only receives the tip if every preceding instruction succeeded. If the tip came first and a later instruction failed, the tip would be paid for a bundle that had no effect.

The transaction is compiled as a Version 0 message (v0::Message::try_compile), signed by the payer (VersionedTransaction::try_new), serialized with bincode::serialize, and base64-encoded for the JSON-RPC body. An application integration passes its own Vec<Instruction> as the payload; the rest of the pipeline is identical.

2.3 The UUID Requirement and Discovery

What the documentation said. Jito's documentation described the x-jito-auth UUID as a rate-limiting mechanism. Without it, users submit at a lower rate limit or are throttled after a threshold, but are otherwise treated normally.

What actually happens. When the x-jito-auth header is absent or carries an invalid UUID, the Block Engine:

1. Returns HTTP 200 with a well-formed JSON-RPC success response.
2. The result field contains a valid 64-character hex bundle ID, indistinguishable from a legitimately accepted bundle.

3. The bundle is never forwarded to any validator. It is silently discarded.
4. A subsequent call to `getInflightBundleStatuses` returns `"status": "Invalid"`.

This is not rate limiting. It is access control, and the HTTP response gives no indication of it.

The investigation. Runs 1 through 5 had `submitted_slot = 0` (the Geyser stream had not yet connected) and were consistent with a startup race condition. Runs 6 through 15 had valid slot numbers, connected streams, and tips ranging from 4,973 to 50,000 lamports. Run 12 used a 50,000 lamport tip, well above any plausible p95 at normal conditions. All 15 received valid bundle IDs. None landed. Querying `getInflightBundleStatuses` for run 12's bundle ID returned `Invalid`. The same was true for all 15.

No tip adjustment, blockhash refresh, endpoint change, or encoding variation resolved the failures. The only variable that mattered was whether `COPILOT_JITO_UUID` was set. Run 16 was the first submission with the UUID configured. It landed in 705 milliseconds.

What this means architecturally. HTTP 200 plus a bundle ID carries no signal about whether the bundle will land. The only meaningful confirmation is a landing event in the Geyser stream. This is one of the core reasons lifecycle tracking is stream-based rather than API-based: the stream is the only source of a definitive, timely landing signal.
Any submission infrastructure that does not monitor the stream and does not check `getInflightBundleStatuses` will silently "succeed" on every attempt while landing zero bundles. Fifteen mainnet runs confirm this failure mode is not theoretical.

2.4 Lifecycle Tracking and Commitment Levels

Solana uses three commitment levels with distinct guarantees:

- Processed: The local validator has received and executed the block. No voting has occurred yet. In practice on a healthy mainnet cluster, processed transactions almost never revert. This is the first signal the lifecycle tracker detects.
- Confirmed: The block has received votes from more than two-thirds of stake-weighted validators. Tower BFT lockout rules make reverting at this point prohibitively expensive. Confirmed lags processed by roughly 1 to 2 slots (400 to 800 milliseconds).
- Finalized: The block has accumulated 32 consecutive Tower BFT confirmation votes. It is irreversible. Finalized lags confirmed by approximately 31 slots (12 to 14 seconds at 400 ms/slot).

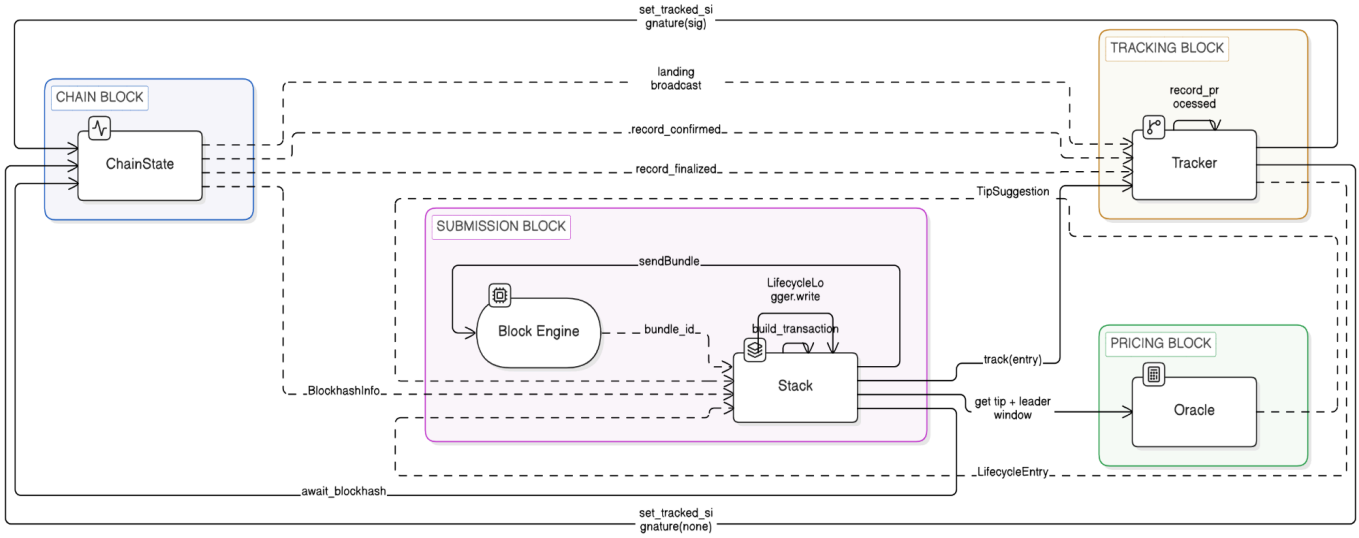
The `SignatureTracker` watches for landings by installing a filter on the Geyser stream (`set_tracked_signature()` writes to a `watch` channel that the geyser task reads), then

listening on a broadcast channel for the landing slot. Confirmed and finalized are detected by comparing the shared ChainState slot counters against the landing slot, with no network calls. When the confirmed counter reaches or passes the landing slot, the transaction is confirmed. When the finalized counter does, it is finalized.

The landing deadline is 90 seconds (DEFAULT_LANDING_DEADLINE), calibrated against the 150-slot blockhash lifetime (~60 seconds) with a 30-second buffer. For fault injection, the deadline is reduced to 25 seconds because the corrupted blockhash guarantees failure, and waiting the full 90 seconds would add unnecessary delay before the agent decision.

2.5 The Full Submission Lifecycle

The diagram below shows a single successful submission from invocation through finalization.



Timing observations from all thirteen confirmed mainnet landings:

Run	Tip (lamports)	Submitted slot	Landed slot	Submit to processed	Processed to confirmed	Confirmed to finalized
16	5,040	427,324,339	427,324,344	705 ms	590 ms	11,836 ms
17	10,479	427,324,380	427,324,383	467 ms	558 ms	13,651 ms

21	7,933	427,325,285	427,325,291	708 ms	840 ms	12,794 ms
23	100,000	427,325,740	427,325,744	412 ms	511 ms	12,328 ms
27	5,000	427,360,303	427,360,306	391 ms	703 ms	11,885 ms
28	5,876	427,955,974	427,955,982	442 ms	465 ms	12,625 ms
29	4,088	427,956,018	427,956,020	461 ms	719 ms	11,422 ms
35	4,744	427,957,277	427,957,282	385 ms	469 ms	12,018 ms
37	5,205	427,957,581	427,957,586	521 ms	213 ms	12,233 ms
41	2,987	427,958,384	427,958,388	457 ms	453 ms	12,249 ms
48	3,460	427,959,929	427,959,936	1,683 ms	467 ms	11,706 ms
49	3,460	427,959,971	427,959,974	583 ms	704 ms	11,791 ms
56	7,268	427,961,210	427,961,212	318 ms	462 ms	12,079 ms

Submit-to-processed (318 to 1,683 ms, with run-48 as an outlier, all other new landings: 385 to 583 ms) reflects Block Engine routing time and where in the Jito leader's four-slot window the submission arrived. Processed-to-confirmed (213 to 840 ms across all thirteen landings; all under one second) reflects supermajority vote propagation and

is consistent with a healthy cluster throughout. Confirmed-to-finalized (11.8 to 13.7 seconds) matches the Tower BFT 32-slot lockout at 400 ms/slot. An application needing irreversible confirmation must budget roughly 13 to 15 seconds from submission.

3. Failure Classification

Most Solana submission failures are silent. The Block Engine returns HTTP 200, the bundle gets an ID, and nothing happens. The application must infer what went wrong from observable signals: blockhash age, leader availability, tip level relative to the market, and the Block Engine's post-submission status.

The fault crate encodes this inference as a pure synchronous classifier: same inputs always produce the same output, testable without a runtime.

3.1 Failure kinds

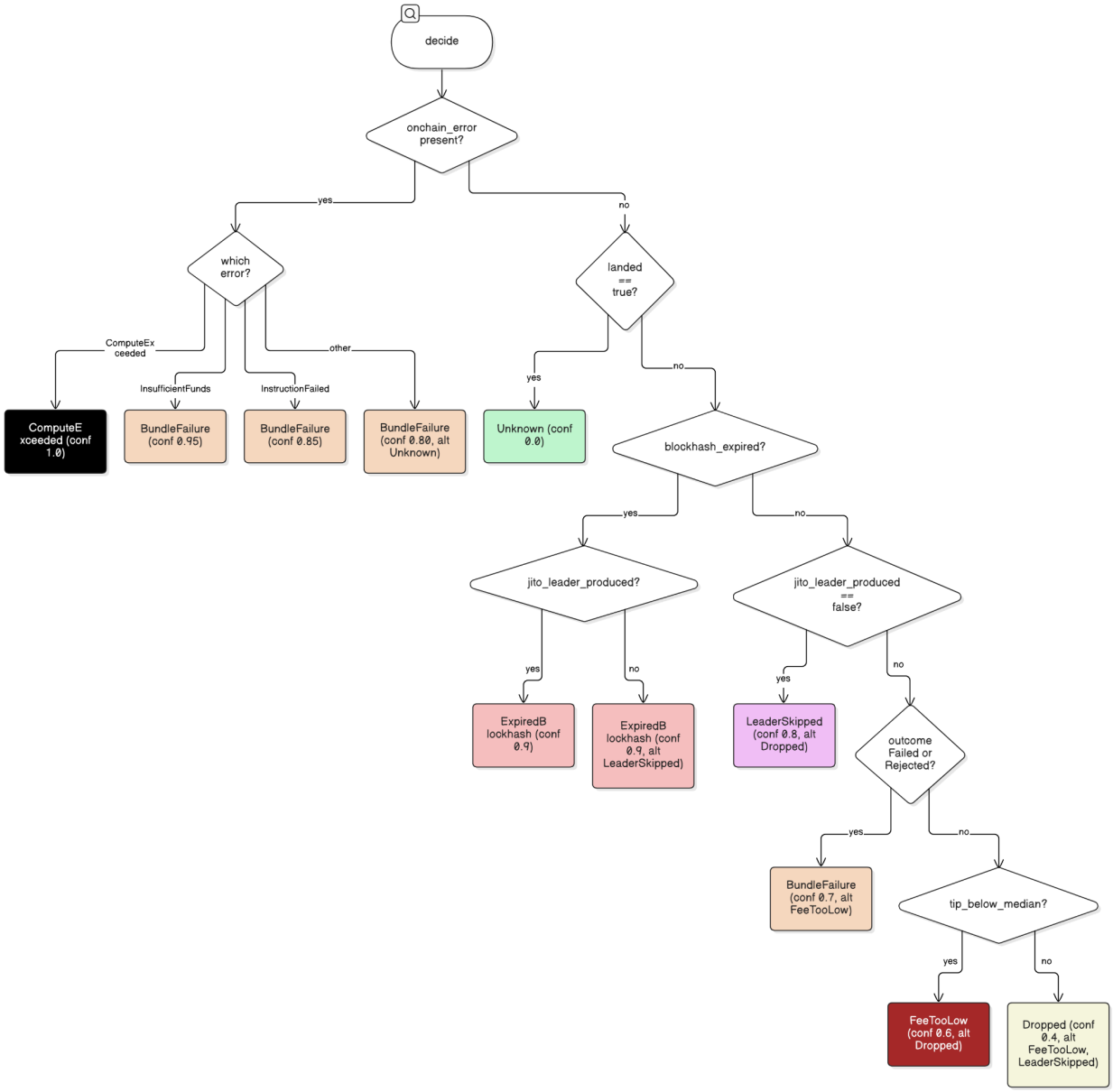
Kind	Meaning	Recovery
ExpiredBlockhash	Blockhash aged past 150 slots before inclusion	Fresh blockhash, hold tip unless congestion also rose
FeeTooLow	Tip was below the recently landed median; bundle was outbid	Raise tip toward or above p75, scaled by congestion
ComputeExceeded	Transaction landed but ran out of compute units	Cannot retry same transaction, fix CU limit upstream
BundleFailure	Block Engine failed or rejected: UUID absent, simulation failure, or onchain instruction error	Varies, UUID absence requires obtaining the UUID
LeaderSkipped	No Jito-connected validator produced during the submission window	Wait briefly, retry with unchanged tip
Dropped	Never landed with no decisive cause; tip looked adequate and leader produced	Conservative retry, modest tip raise is reasonable

Unknown	Transaction landed without error; not a failure	N/A

3.2 Observable signals the classifier uses

Signal	Source	Failure kinds it supports
blockhash_age_slots > max (150)	ExpiredBlockhash.apparent_age_slots or slot delta	ExpiredBlockhash
jito_leader_produced == false	Leader schedule lookup	LeaderSkipped when absent
outcome == Failed or Rejected	getInflightBundleStatuses mapped to SubmissionOutcome	BundleFailure
tip_lamports < recent_landed_tip_p50	Paid tip vs oracle p50	FeeTooLow
onchain_error present	Transaction result from geyser (if landed with error)	ComputeExceeded, BundleFailure subtypes

The following diagram shows the exact evaluation order in fault::classify(). The first matching branch wins.



Each result carries confidence, a rationale string with the specific values that drove the decision, and an alternatives list for kinds that cannot be ruled out. The Dropped fallback at 40 percent confidence is deliberate: manufacturing false certainty would cause the agent to take the wrong recovery action. A LeaderSkipped misdiagnosed as FeeTooLow produces an unnecessary tip increase. During runs 6 through 15, the actual cause was a missing UUID (BundleFailure) with tips as high as 50,000 lamports. Blindly raising the tip would never have resolved it.

Deterministic fault injection. inject_expired_blockhash() takes the current valid blockhash, rotates the bytes left by one position, XORs the first byte with 0xA5, and sets

apparent_age_slots = 166 (16 past the 150-slot window). The result is syntactically valid but is not in any validator's recent blockhash cache. It is guaranteed to fail. The Block Engine still receives a valid bundle and returns a real bundle ID; the failure occurs when the transaction reaches a validator and is checked against the blockhash cache.

4. The Autonomous Retry Agent

After a bundle fails, the question is not just whether to retry but how: at what tip, with what blockhash, and for what reason. Copilot delegates this decision to Claude Sonnet via the Anthropic Messages API. The agent receives the typed failure event plus live market context, returns a structured JSON decision, and the pipeline executes it.

4.1 Separation of concerns

The fault crate classifies deterministically over observable signals. The agent crate reasons over that classification using live chain and market data. Neither can do the other's job. The classifier cannot consider whether the auction has changed since the failure. The agent cannot compute blockhash expiry from raw slot data without the classifier's framework. Together they form a two-stage recovery pipeline.

4.2 The System prompt

The prompt encodes four causal relationships the agent must apply: a blockhash expiry needs a fresh blockhash (not a higher tip, unless congestion also rose); a tip below median needs a higher tip scaled to the congestion read; a skipped leader is transient (wait briefly; tip rarely needs to change); compute exhaustion is not retryable as-is. The prompt also encodes decision calibration: higher confidence in a clear cause warrants a more decisive action; genuine ambiguity warrants a conservative one. The model is constrained to exactly three actions (retry, abort, wait) and must respond with only a JSON object.

Context object: The agent receives four groups of fields:

Group	Contents
failure	kind, confidence, rationale, alternatives, and the full signals struct (blockhash_age_slots, tip_lamports, jito_leader_produced, outcome, recent_landed_tip_p50, onchain_error)
chain	processed_slot, confirmed_slot, finalized_slot at the moment the context was assembled

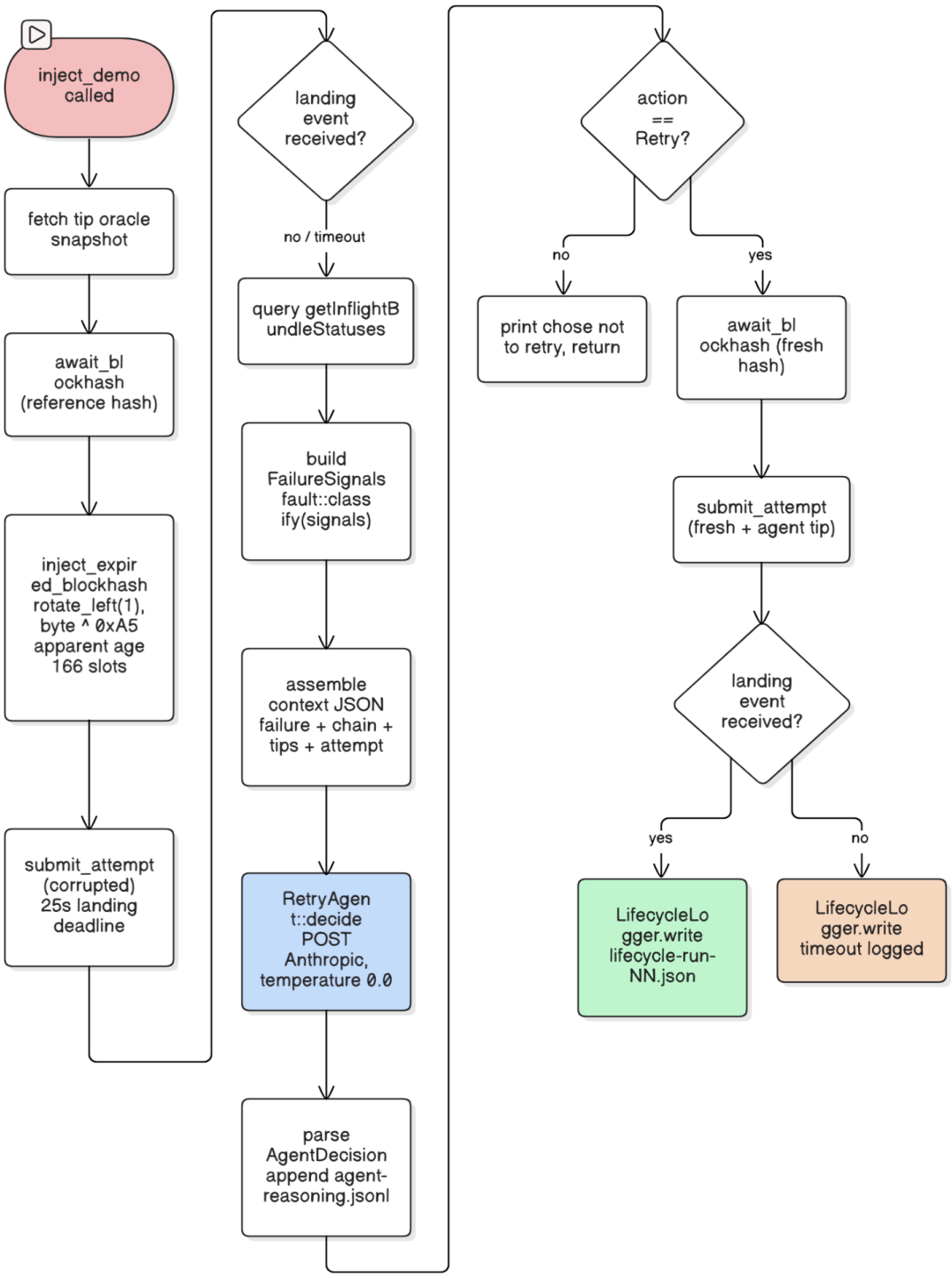
tips	Full TipSuggestion: all six jito_tip_floor fields (p25, p50, p75, p95, p99, ema_p50), prio_fees percentiles and sample count, and congestion (level, tail ratio, median_rising)
failed_attempt	signature, tip_lamports, landed

The agent receives the full oracle snapshot rather than a summary so it can do arithmetic over the percentiles and verify claims rather than trusting a recommendation.

Decision schema:

Field	Type	Meaning
action	"retry", "abort", or "wait"	What the pipeline should do next
new_tip_lamports	integer or null	Tip for the retry; null means hold the current tip
reasoning	string	One or two sentences explaining the decision; logged to agent-reasoning.jsonl
confidence	float (0 to 1)	The model's self-reported confidence in the decision

The retry loop is shown below. The "Pipeline" lane is inject_demo() in pipeline.rs.



The recorded decision. From logs/agent-reasoning.jsonl (four entries total). Entry 1 is from the original inject session (runs 26 and 27). Entries 2 through 4 are from the three additional inject sessions run in the extended test (runs 51/52, 53/54, and 55/56). The decision quoted below is Entry 1, the original:

Context presented to the agent included: kind: "expired_blockhash", confidence: 0.9, rationale: "blockhash aged 166 slots, past the 150-slot window, before inclusion", alternatives: [], tip_lamports: 5000, recent_landed_tip_p50: 1039, jito_tip_floor.p75: 5000, congestion.level: "high", tip_tail_ratio: 84.09817131857555, median_rising: false.

The agent returned:


```
{
  "action": "retry",
  "confidence": 0.9,
  "new_tip_lamports": 5000,
   "reasoning": "The failure is clearly due to an expired blockhash (166 slots > 150-slot window), not a tip issue — the existing tip of 5000 lamports already sits at the p75 level which is sufficient given current congestion. A fresh blockhash is all that's needed to retry successfully."
}
```


The retry (run 27) landed in slot 427,360,306, 391 milliseconds after submission, with the same 5,000 lamport tip.

The fourth inject session (Entry 4, runs 55 and 56) produced a different and equally instructive decision. At the time of submission, the oracle showed congestion.level = 'moderate' with median_rising = false. The initial tip was 7,268 lamports (at p75). The agent returned:


```
{
  "action": "retry",
  "confidence": 0.92,
  "new_tip_lamports": null,
   "reasoning": "The failure is clearly due to an expired blockhash (166 slots > 150-slot window), with 0.9 confidence; the fix is simply fresh blockhash. The current tip of 7,268 lamports is already at the p75 landed level, which is adequate given moderate (non-rising) congestion, so no tip adjustment is needed."
}
```


new_tip_lamports = null means hold the tip at 7,268 lamports. Run 56 landed in slot 427,961,212, 318 milliseconds after submission. Entries 2 and 3 (runs 51/52 and 53/54) are also in the log; in each, the agent issued a retry decision with a tip adjustment calibrated to the oracle snapshot at that moment. Those retries did not land within the

deadline (90 seconds for regular submissions; 25 seconds for injected faults). All four decisions are recorded in `logs/agent-reasoning.jsonl`.

4.3 Why is this real reasoning and not a lookup table?

A hardcoded policy would say: on `ExpiredBlockhash`, hold the tip. The agent does something structurally different, it reads `jito_tip_floor.p75 = 5000` from the oracle data, compares it to the paid tip of 5,000, and concludes that the tip is adequate. The statement "already sits at the p75 level" is not a rule; it is a computation over live data that happened to produce a hold decision.

The distinction matters in cases the hardcoded rule cannot handle. If the blockhash had expired and the auction had simultaneously moved (say, p50 rising to 80,000 lamports and congestion reaching `Severe`), a hold-tip rule would bid 5,000 into an 80,000-lamport auction. The agent, given the same structure of context with different oracle values, would see the gap and raise the tip accordingly, while still refreshing the blockhash for the same reason. The decision is data-dependent, not rule-bound.

5. Design Decisions and Operational Evidence

5.1 Key Infrastructure Decisions

Decision	Alternative rejected	Why the chosen approach wins
Persistent Yellowstone gRPC stream	RPC polling	Polling cannot support signature subscription; stream delivers sub-slot latency and all three commitment levels on one connection
`Arc<ChainState>` with lock-free atomics	Channel-based actor with message passing	Lock-free reads (AtomicU64, ArcSwapOption) eliminate contention for a small, frequently-read, monotonically-updated state object

Health-aware exponential backoff (reset after any message received)	Fixed reconnect interval	The Geyser provider periodically sends clean closes during quiet periods; a fixed interval would accumulate delays unnecessarily
fault crate with no async I/O	Async classifier that queries the API to fill missing signals	Synchronous classifier is deterministic, testable without a runtime, and decouples signal collection (pipeline's responsibility) from classification logic
Honest confidence scores with alternatives	Single best-guess FailureKind with no uncertainty	Misdiagnosed failures produce wrong recovery actions; Dropped at 40 percent with alternatives is more useful than FeeTooLow at high false confidence
Request-time tip pricing via tokio::try_join!	Startup-cached tip values	The Jito auction changes continuously; a cached value from the start of a session can be wrong within minutes

The most consequential decision is the streaming architecture. Every other component depends on the freshness and completeness of the data in ChainState. The fault crate's design (no async, pure function) is the second most important: it keeps the critical classification path free from network dependencies and timing variability.

5.2 Confirmed Landings

All thirteen confirmed landings occurred after COPILOT_JITO_UUID was configured (runs 16 onward). Data pulled directly from logs/success/:

Run	Bundle ID prefix	Tip (lamports)	Submitted slot	Landed slot	Submit to processed	Processed to confirmed	Confirmed to finalized

16	9b3e0749	5,040	427,324,339	427,324,344	705 ms	590 ms	11,836 ms
17	8316828c	10,479	427,324,380	427,324,383	467 ms	558 ms	13,651 ms
21	b5b86d79	7,933	427,325,285	427,325,291	708 ms	840 ms	12,794 ms
23	ecda7af4	100,000	427,325,740	427,325,744	412 ms	511 ms	12,328 ms
27	0e92060a	5,000	427,360,303	427,360,306	391 ms	703 ms	11,885 ms
28	c87d453a	5,876	427,955,974	427,955,982	442 ms	465 ms	12,625 ms
29	c56ab240	4,088	427,956,018	427,956,020	461 ms	719 ms	11,422 ms
35	d7a62fcd	4,744	427,957,277	427,957,282	385 ms	469 ms	12,018 ms
37	b243e97d	5,205	427,957,581	427,957,586	521 ms	213 ms	12,233 ms
41	a485dba5	2,987	427,958,384	427,958,388	457 ms	453 ms	12,249 ms
48	f7ea5b61	3,460	427,959,929	427,959,936	1,683 ms	467 ms	11,706 ms
49	665fcfac	3,460	427,959,971	427,959,974	583 ms	704 ms	11,791 ms

56	6991b55b	7,268	427,961,210	427,961,212	318 ms	462 ms	12,079 ms

Runs 16, 17, 21, 28, 29, 35, 37, 41, 48, and 49 used oracle-priced tipsat p75. Run 23 used a manually pinned 100,000 lamport tip. Runs 27 and 56 were agent-directed retries; run 27 held the tip at p75 per the agent's decision, and run 56 also held at p75 (7,268 lamports) under moderate congestion with a non-rising median. There is no strong correlation between tip amount and landing speed across the full dataset: run 41 landed in 457 ms with a 2,987 lamport tip; run 48 had an anomalously high 1,683 ms submit-to-processed delay unrelated to tip level. The variable that determines landing speed is Jito leader window timing, not tip competitiveness.

5.3 What the Processed-to-Confirmed Delta Reveals

The processed-to-confirmed delta measures the time for a supermajority of stake-weighted validators to receive, verify, and vote on a block. For all thirteen landings, the deltas were: 590, 558, 840, 511, 703, 465, 719, 469, 213, 453, 467, 704, and 462 ms. All are under one second. The distribution across two separate testing sessions (runs 16-27 and runs 28-56) shows the same sub-second supermajority vote propagation pattern, confirming cluster health was consistent across both sessions.

A healthy mainnet cluster shows processed-to-confirmed deltas in the 400 to 800 ms range. Consistent deltas above 1.5 seconds indicate vote propagation slowdown from validator network issues, high vote load, or fork resolution. The consistency across sessions with different tips and different session times confirms that the cluster was healthy throughout all thirteen recording windows.

5.4 The Fault Injection Run

This part covers the original inject session (runs 26 and 27). Three additional inject sessions were conducted in the extended test, those are covered below.

Run 26: `submitted_slot = 427,360,222`, `tip_lamports = 5,000`, `failure = "ExpiredBlockhash"`, all commitment timestamps null.

Classifier inputs: `blockhash_age_slots = 166`, `max_blockhash_age_slots = 150`, `jito_leader_produced = true`, `outcome = "failed"` (Block Engine returned Invalid, mapped to Rejected), `tip_lamports = 5,000`, `recent_landed_tip_p50 = 1,039`.

Classifier output: `FailureKind::ExpiredBlockhash`, confidence 0.9, rationale "blockhash aged 166 slots, past the 150-slot window, before inclusion", alternatives `[]`. The

alternatives list is empty because `jito_leader_produced = true` eliminates `LeaderSkipped`.

Agent decision: `action = "retry"`, `new_tip_lamports = 5000`, confidence 0.9.

Run 27 landed in slot 427,360,306 with the agent-held 5,000 lamport tip, 391 ms after submission. The gap between run 26's submitted_slot (427,360,222) and run 27's (427,360,303) was 81 slots, approximately 32 seconds, covering the 25-second landing deadline, the Anthropic API call, and the blockhash re-fetch.

Extended Tests

Three more `copilot inject` runs were performed in the extended test session.

Session 2: Runs 51 and 52

Run 51: submitted_slot = 427,960,281, tip_lamports = 1,806, failure = "ExpiredBlockhash", all commitment timestamps null. The oracle showed congestion level = "moderate" with median_rising = true. Agent decision: action = "retry", new_tip_lamports = 2,000, confidence = 0.92. Reasoning: "The failure was caused by an expired blockhash (166 slots > 150-slot limit), so a fresh blockhash is the primary fix. Congestion is moderate with a rising median, so a modest tip increase from 1806 to 2000 (slightly above p75) is prudent to ensure landing." Run 52 did not land within the deadline.

Session 3: Runs 53 and 54

Run 53: submitted_slot = 427,960,765, tip_lamports = 32,500, failure = "ExpiredBlockhash", all commitment timestamps null. The oracle showed congestion level = "severe" with median_rising = true. Agent decision: action = "retry", new_tip_lamports = 50,000, confidence = 0.88. Reasoning: "The failure is clearly due to an expired blockhash (166 slots > 150 slot limit), so a fresh blockhash is required. Congestion is severe with a rising median, and the current tip of 32500 sits exactly at p75; bumping slightly above p75 improves landing probability under severe congestion." Run 54 did not land within the deadline.

Session 4: Runs 55 and 56

Run 55: submitted_slot = 427,961,131, tip_lamports = 7,268, failure = "ExpiredBlockhash", all commitment timestamps null. The oracle showed congestion level = "moderate" with median_rising = false. Agent decision: action = "retry", new_tip_lamports = null (hold at 7,268), confidence = 0.92. Reasoning: "The failure is clearly due to an expired blockhash (166 slots > 150-slot window) with 0.9 confidence; the fix is simply a fresh blockhash. The current tip of 7,268 lamports is already at the p75 landed level, which is adequate given moderate (non-rising) congestion, so no tip

adjustment is needed." Run 56 landed in slot 427,961,212, 318 ms after submission, with the agent-held 7,268 lamport tip.

What the four sessions show together:
Across four sessions the classifier returned the same result every time: kind = ExpiredBlockhash, confidence = 0.9, alternatives = []. The agent's tip decision varied in each case because the oracle snapshot varied. When congestion was moderate and non-rising, the agent held the existing tip. When congestion was moderate and rising, the agent raised the tip modestly. When congestion was severe, the agent raised the tip significantly. This is data-driven reasoning over live oracle values, not a hardcoded rule.

Landed runs are stored in `logs/success/`, non-landing and injected-fault runs in `logs/failures/`, and `agent-reasoning.jsonl` at the logs root.

5.5 Jito Leader Misses

Run 24 is the clearest leader-window miss in the dataset. Runs 23 and 24 were submitted 48 slots apart with identical 100,000 lamport tips.

- Run 23: `submitted_slot = 427,325,740`, `landed_slot = 427,325,744` (412 ms)
- Run 24: `submitted_slot = 427,325,788`, `landed_slot = null`

At slot 427,325,740, a Jito-connected validator held the leader slots immediately following submission. By slot 427,325,788, that window had closed. The next 256 slots contained no Jito-connected validator in the lookahead. The Block Engine held the bundle, but every validator producing blocks in that window was not connected to the engine. The bundle expired without landing.

A 100,000 lamport tip placed run 24 above p99. Fee competitiveness was not a factor. The failure was entirely a routing problem. An operator without leader-schedule awareness would likely raise the tip in response. Copilot's pre-submission leader window log provides the context to make the correct diagnosis at submission time.

The separate log messages "Jito leader set unavailable; Block Engine will route" and "no Jito leader within lookahead; Block Engine will route" distinguish two conditions: the first means `getConnectedLeaders` failed or returned nothing (the Block Engine's routing intelligence is the only fallback), and the second means the Jito set was fetched successfully but no connected validator appears in the next 256 slots. In both cases the bundle is submitted, but neither case offers the predictable, fast landing that a known Jito window provides.