The CodeRunner Queue

On 24 August, one customer's automation hitting its own API rate limit took our server out of service three times. This explains the fix we are building: a waiting line for customer scripts, how it survives every kind of restart, and what it does and doesn't promise.

Status · Designed, not yet built Audience · Non-technical 28 August 2026

What CodeRunner is, in one paragraph

Customers write small scripts that run automatically when something happens on their Businessmap board — a card moves, a deadline passes, a field changes. A business rule fires, our server runs the customer's script, and the script calls the Businessmap API to do whatever it was written to do: create a card, post a comment, update a field. About **420 of these run every day**. A typical one finishes in **1.6 seconds**, and 97% finish inside five.

That is the healthy case, and it is the overwhelming majority. The problem is what happens to the unhealthy few, and how much damage they can do on the way.

What went wrong on 24 August

Every Businessmap account has an API rate limit. When a script exceeds it, the API replies "you're going too fast — try again in 90 seconds." Our code takes that literally: it **waits 90 seconds and tries again**, and it will do that indefinitely.

Waiting costs no processing power, so nothing looks busy. But the script is still holding one of the server's limited number of *worker slots* — think of them as service counters. A waiting script occupies a counter and does nothing with it.

Three times that day, one account's scripts filled almost every counter with scripts that were only waiting. And those counters are shared: the same pool serves the customer portal, the developer portal, and the automated health check that tells our load balancer whether this server is alive.

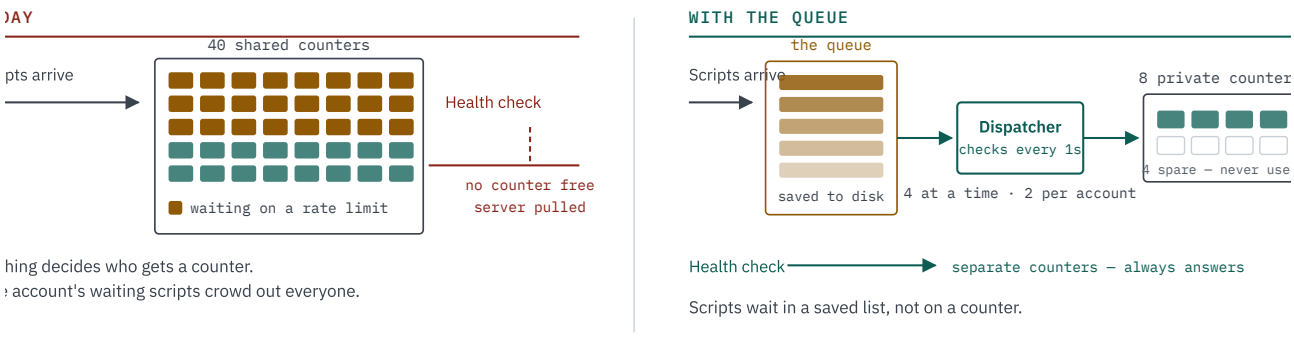
46% of executions in the 12:08 window died at the five-minute cut-off — 143 of 309	**24 / 40** counters held continuously by scripts that were never going to succeed	**191 s** for the health check to answer, against a normal figure of milliseconds
3× times that day the load balancer pulled the server out of service		

A single customer's rate limit took the box offline. Not through malice and not through a bug in their script — simply because nothing in the system decided how many scripts were allowed to run at once.

> THE PART THAT IS EASY TO MISS
>
> The crowding was also *causing* the rate limits. Rate limits are counted per API key, and at peak those scripts were all competing against the same key's allowance. Fewer running at once means fewer rate limits, which means fewer waiting scripts. The problem feeds itself, and so does the fix.

The fix, in one picture

Nothing today decides who gets a counter. A script arrives, takes one, and holds it until it finishes or is cut off at five minutes. We are adding the thing every service desk has and our system doesn't: **a waiting line, and someone deciding who is called next.**



e queue adds one component and one decision. Waiting now happens in a list saved to disk, ıere it costs nothing, instead of on a service counter, where it costs everything. The dispatcher eases at most four scripts at a time and at most two from any one account. CodeRunner also has its 'n eight counters now, separate from the portal and the health check — that part already shipped, d it is a safety net rather than the policy: if the dispatcher is working, four of those eight are never ıched.

Three states, and why the difference matters

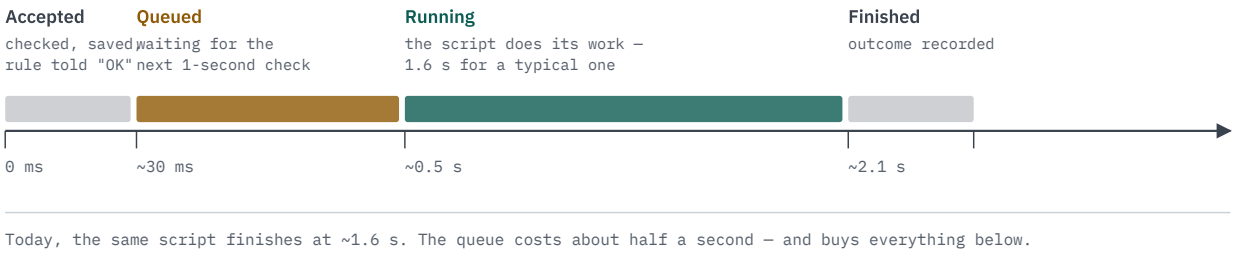
Every execution is in exactly one of three states, and the whole design turns on keeping them distinguishable. Today the system cannot reliably tell the second from the third, which is why a backlog and an outage look the same in the logs.

● QUEUED	Accepted, written down, waiting its turn. Nothing has been attempted yet, so nothing has changed on the customer's board. This state is new — today it does not exist.
● RUNNING	A counter is doing the work right now. Cards may already have been created, comments already posted.
● FINISHED	Done, successfully or not. If not, the reason is recorded.

> WHY SUPPORT CARES ABOUT THIS
>
> A queued script is **late**. A failed script is **lost**. If the log shows the same thing for both, then a two-minute backlog during a busy hour reads as an outage, and support escalates something that is working as designed. Making "queued" visible and distinct is a requirement of this project, not a nice-to-have.

The life of one script

A customer's card moves. Here is what happens, second by second, in the ordinary case.



Today, the same script finishes at ~1.6 s. The queue costs about half a second — and buys everything below.

e ordinary case, end to end. The business rule still gets its instant "OK" and never waits — that is changed. The only new step is the pause between being written down and being picked up, which erages half a second because the dispatcher checks once per second.

> UNDER LOAD
>
> The busiest minute we have ever recorded had 137 scripts arrive at once. Four running at a time, at 1.6 seconds each, clears roughly 2.5 per second — so that entire burst drains in **under a minute**, with the last script in the line waiting about 55 seconds. Today that same burst is what took the server down.

The dispatcher: a service that checks once a second

The dispatcher is a small program that runs continuously on the server. Its whole job is a loop: look at the list, see what is eligible to start, start it, repeat. It does not run the customer's script itself — it hands the work to a counter and moves on. That separation is deliberate, and the reason appears in the failure table below.

Why once a second, and what it costs

Checking once a second means a script waits half a second on average before it is noticed. The check itself is a single indexed database lookup that usually returns nothing: about a quarter of a millisecond. Across a whole day that is **roughly thirty seconds of database time** — about three hundredths of one percent of one processor. Against an average arrival rate of seventeen scripts an hour, the polling is not a meaningful cost.

Why it is not scheduled, but always running

The obvious alternative is a scheduled task that starts every minute. We rejected it: a task that starts every minute leaves a gap at each minute boundary, and it exists mainly to avoid a problem — a database connection going stale from disuse — that cannot occur when the connection is used every

single second.

Instead the dispatcher is registered with `systemd`, the standard Linux service manager, the same way our existing monitoring service is. That gives it three properties that matter, and they are the answer to most of the next section.

- **It starts automatically when the server boots.** Nobody has to remember.
- **If it crashes, it is restarted within one second.** Not by anyone noticing — by the operating system, unconditionally, forever.
- **Only one copy can run.** A second copy would double-run scripts; a lock file makes that impossible even if someone starts one by hand.

What happens when things break

This is the question the design has to answer well, so it is worth being precise about it. The single most important fact is this:

> THE RULE THAT MAKES THE REST WORK
>
> **The queue is rows in the database, not a list held in memory.** A script that has been accepted is written to disk before the business rule is even told "OK". Nothing in memory — not in the dispatcher, not in the web server — has to survive for that work to survive. Anything that restarts comes back, reads the same list, and carries on from where the list says it is.

ERE STATE LIVES, AND WHAT A RESTART DOES TO IT

y the first box has to be durable — and it is the only one that is.

o of the three stores are allowed to vanish. The cooldowns and the running-counts are nveniences: losing them costs a few seconds of slightly worse decisions, not any work. Everything at must not be lost lives in one place, and that place is the database.

BREAKS	WHAT ACTUALLY HAPPENS	WORK LOST	BACK TO NORMAL
ispatcher crashes	systemd restarts it. It rereads the list and continues. Scripts that were already running are unaffected — the dispatcher does not run them, it only starts them.	None	~1 second

BREAKS	WHAT ACTUALLY HAPPENS	WORK LOST	BACK TO NORMAL
ispatcher stops nding without ing	The honest weak point. Nothing is lost — the queue simply stops draining, and scripts pile up as `● QUEUED`. This is why we monitor how long the oldest waiting script has been waiting: a stalled dispatcher and an idle one look identical on a count alone, and must not.	None	Manual restart
estarts, or we deploy	Scripts that were mid-run are killed and marked failed, exactly as today. Everything still waiting is untouched and starts flowing again as soon as PHP is back.	In-flight only	Seconds
ıe restarts	Same as above. Apache is only the door the dispatcher knocks on; the queue is behind it, not in it.	In-flight only	Seconds
atabase restarts or nes unreachable	The dispatcher's connection drops; it reconnects and resumes. No queued work is lost, because the queue *is* the database. But new scripts cannot be accepted while it is down — a business rule firing in that window gets an error rather than being silently swallowed. That is true of CodeRunner today too; the queue neither improves nor worsens it.	New arrivals only	Seconds after DB ret
hared cache restarts	Rate-limit cooldowns are forgotten, so the first script for each affected key runs, gets a rate-limit reply, and the cooldown is immediately relearned. Slightly worse decisions for a few seconds.	None	Immediate
/hole server reboots	All of the above at once. systemd starts the dispatcher as part of boot; it finds the same list and drains it. Nothing needs a human.	In-flight only	Boot time
ly maintenance, ; UTC	Automatic updates restart services in that window, so treat it as a scheduled version of the row above. Worth noting: **the busiest hour we have ever recorded falls inside this window**, so the acceptance test for this project has to be a hard kill during maintenance, not a polite shutdown.	In-flight only	Seconds
pt's own run is cut off e minutes	Marked failed, with a reason, and **not retried**. See below — this is a deliberate choice, not a gap.	That script	n/a

> **WHY WE NEVER AUTOMATICALLY RETRY**
>
> A CodeRunner script has usually already *done* things by the time it dies — created three cards, posted two comments, moved a card to another column. Re-running it from the beginning would do all of that a second time. A customer would rather have one incomplete automation they can see in the log than two sets of duplicate cards on their board they have to clean up by hand. So a script that dies stays dead, and says so.
>
> This is unchanged from today, and a queue makes it tempting to change. It should not be changed without solving the duplication problem first.

How API rate limits are handled

This is where the biggest customer-visible improvement is, and it is worth spelling out because the current behaviour is genuinely poor in one specific case.

Rate limits come in two kinds. A **per-minute** limit says "wait about ninety seconds." An **hourly** limit says "wait about thirty minutes." Today the system treats these completely differently, and both treatments are bad.

TODAY · PER-MINUTE LIMIT

Sits and waits, forever

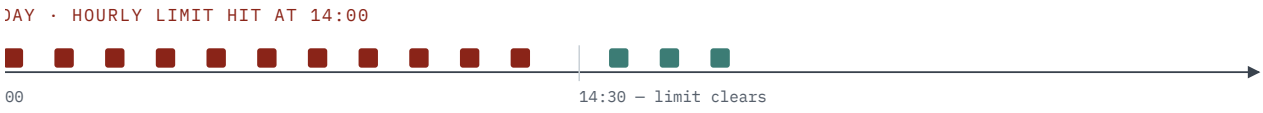
The script holds its counter, sleeps ninety seconds, tries again — and again, with no limit on the number of attempts. It is finally cut off at five minutes, having achieved nothing and blocked a counter the whole time.

TODAY · HOURLY LIMIT

Gives up immediately

The script fails on the spot. During an hourly limit, *every* execution for that account fails for as long as the limit lasts — potentially hundreds of automations silently not happening.

The queue lets us replace both with one mechanism. When the API says "not until 14:30", we write that down against that API key, and the dispatcher simply **does not start** that key's scripts until 14:30. They stay `● QUEUED`. Nobody holds a counter, nothing fails, and every other customer is completely unaffected.



Every execution in the window fails. The work never happens, and nobody is told.

Every execution is held, then runs. Late instead of lost — and no counter was held meanwhile.

e hourly-limit case is the clearest win. The same automations that silently do not happen today e simply delayed and then done. This only becomes possible once there is a queue: without newhere to park the work, giving up really was the only option available to the old code.

WHAT THIS DOES NOT SOLVE

Holding scripts back protects the ones that *haven't started*. It cannot help a script that is already halfway through when the limit is hit — that one has already created cards, so it can neither be safely restarted nor cleanly abandoned. For those we will keep a short wait-and-retry, but capped at three attempts instead of unlimited.

The cooldown is also a good guess rather than a measurement: the same API allowance is shared with the customer portal, our other tools, and the customer's own integrations. We will still occasionally be rate-limited by traffic we did not generate.

Why one customer can no longer crowd out another

Two separate limits do two separate jobs, and it is worth keeping them apart because they are counted differently.

FAIRNESS · COUNTED PER ACCOUNT

At most 2 running per account

Three accounts generate 89% of all CodeRunner traffic, and one of them writes scripts that take minutes rather than seconds. Without a per-account limit, a single fast-moving customer would sit behind a slow one's queue and their automation would be minutes late through no fault of their own.

QUOTA · COUNTED PER API KEY

No starts while a key is paused

Rate limits are charged against the API key, not the account, so the cooldown is tracked per key. If an account uses several keys, the per-account limit above still stops it taking every slot.

Above both sits the hard ceiling that already shipped: CodeRunner has **eight counters of its own**, separate from the portal and the health check. If the dispatcher is working correctly only four are ever in use. The other four exist so that a bug in this new code — or someone bypassing it entirely — still cannot reach the systems that were taken down in August.

What we promise, and what we don't

WE PROMISE

- An accepted script is **never lost to a restart** of any component, planned or not.
- A rate-limited account **delays only itself**. Every other customer runs at normal speed.
- Customer automations **can no longer take the server offline**. That path is closed twice over — by the queue, and by the separate counters beneath it.
- A late script is **visibly late**, not reported as a failure.
- Typical end-to-end time stays around **two seconds**.

WE DON'T PROMISE

- **No automatic retry.** A script killed mid-run stays failed, deliberately — see the duplication problem above.
- **Not instant.** The queue adds about half a second normally, and can add much longer when an account is rate-limited — though today that same case is a failure rather than a delay.
- **Not immune to a database outage.** New scripts cannot be accepted while the database is down. That is true today as well.
- **Not self-healing against a hung dispatcher.** A crash is handled automatically; a freeze needs monitoring to catch, which is why "how long has the oldest script waited" is a required measurement.

How we will know it works

These are the tests the work has to pass before we would call it done. They are written as things to demonstrate, not things to assert.

- Fire 500 scripts from one account in quick succession. Never more than two run at once, and a second account's script submitted in the middle is delayed by no more than one script's normal run time.
- Kill the dispatcher outright, mid-backlog — not a graceful stop. **Nothing is lost**: everything still waiting is still waiting, and drains when it comes back.
- Put an account under an hourly limit. Its scripts stay queued and then run, and no other account slows down.
- The eight-counter safety net is never reached. If it is, that is a bug in the queue, and we would treat it as one.
- Replay the actual arrival pattern from 24 August, 12:08. The portal and the health check show no effect at all.

– Queue depth and oldest-waiting-age are both visible on a dashboard, so a stalled dispatcher, a busy one and an idle one are three distinguishable things.

How it gets built

Five steps. Everything before the fourth is reversible on its own, and the first is worth having even if the rest were never built.

1. **Cap the retry loops.** Limit the endless wait-and-retry to three attempts. On its own, this plus the separate counters already shipped is a complete fix for the server going offline — the 143 executions that held counters for five minutes each would have released them in about thirty-five seconds. No database changes, nothing else depends on it.
2. **Prepare the records.** Add the fields that let a row say "queued" as distinct from "running", and fix the cleanup job to understand the new state. No behaviour change yet.
3. **Start recording rate-limit cooldowns.** Written down but not yet acted on, so we can confirm they look right before anything depends on them. No risk.
4. **Turn on the dispatcher.** The step that changes behaviour: scripts are accepted into the queue rather than started immediately, and the dispatcher releases them. Includes the safety limits on how deep the queue may get.
5. **Watch and tune.** The numbers chosen here — four at a time, two per account, one-second checks — are sized from fourteen days of real measurements with generous headroom, but they are settings, not architecture. They can be adjusted without rebuilding anything.

Every figure in this document comes from the fourteen days of CodeRunner execution records to 26 August 2026, and from the server logs of the 24 August incident.

The full technical design, including the parts deliberately left out of this summary, is `docs/CODERUNNER_QUEUE.md` in the Solutions & Automations repository.