CHECKLIST

ASP.NET Core API Rate Limiting and Throttling Checklist

A practical checklist to evaluate, configure, test, and roll out ASP.NET Core rate limiting and throttling policies before production.

ASP.NET Core API Rate Limiting and Throttling Checklist

Use this checklist to plan and validate rate limiting in an ASP.NET Core API. It is written for production use and focuses on operational clarity, not just framework setup.

---

1) Define the problem you are solving

- [ ] Identify the resource at risk: database connections, CPU, downstream APIs, queues, file processing, or overall request capacity.
- [ ] State the primary goal: fairness, overload protection, abuse reduction, retry storm control, or tenant isolation.
- [ ] Confirm whether the risk is high request volume, long-running requests, or concurrent work saturation.
- [ ] Decide whether a rate limit, concurrency limit, or layered approach is the best fit.
- [ ] Document what success looks like in measurable terms.

2) Choose the right scope

- [ ] Decide whether the policy should be global or route-specific.

- [ ] Identify endpoints that are more expensive than others.
- [ ] Group routes by work profile instead of applying a single blanket policy everywhere.
- [ ] Confirm whether public, partner, internal, and admin APIs need different treatment.
- [ ] Verify that sensitive or costly endpoints have tighter controls than lightweight endpoints.

3) Choose the partition key

- [ ] Decide what shares a quota: client, user, tenant, API key, IP address, or route.
- [ ] Prefer authenticated identity when reliable authentication is available.
- [ ] Avoid relying only on IP address when traffic may come through NAT, proxies, or shared egress.
- [ ] Confirm that the chosen key matches the actual operational boundary.
- [ ] Validate that one noisy caller cannot consume the quota intended for unrelated callers.

4) Select the limiting algorithm

- [ ] Use a fixed window only if simplicity is the main goal and burst edge effects are acceptable.
- [ ] Use a sliding window if you need smoother behavior near time boundaries.
- [ ] Use a token bucket if short bursts should be allowed while maintaining a long-term average.
- [ ] Use concurrency limiting if backend saturation is caused by too many in-flight requests.
- [ ] Document why the selected algorithm matches the real risk.

5) Define policy values clearly

- [ ] Set a sustained rate that protects shared capacity.
- [ ] Set a burst allowance only if the system can safely absorb it.
- [ ] If using concurrency limits, define the maximum number of simultaneous requests.
- [ ] Choose whether requests will queue or be rejected immediately when capacity is exhausted.
- [ ] Establish limits separately for especially expensive operations when needed.

6) Define rejection behavior

- [ ] Confirm that rejected requests return a clear status code, such as HTTP 429.
- [ ] Decide whether clients receive retry guidance.
- [ ] If retry guidance is provided, make it consistent and meaningful.
- [ ] Ensure rejection responses are part of the API contract.
- [ ] Confirm that rejected requests fail fast instead of consuming unnecessary resources.

7) Plan client behavior

- [ ] Document whether clients should retry after throttling.
- [ ] Define whether retries should use backoff and jitter.
- [ ] Make sure clients understand if the limit is temporary or structural.
- [ ] Verify that automated jobs, batch workers, and dashboards do not retry aggressively.
- [ ] Ensure client teams know which endpoints are subject to stricter policies.

8) Align with upstream infrastructure

- [ ] Review load balancer, reverse proxy, gateway, and container platform limits.
- [ ] Confirm application-layer limits do not conflict with infrastructure-layer protections.

- [ ] Make sure broad traffic filtering happens as early as practical.
- [ ] Use application-layer throttling for richer context and policy precision.
- [ ] Document how layered controls complement each other.

9) Validate behavior under real conditions

- [ ] Test normal traffic patterns first.
- [ ] Test burst traffic near the limit boundary.
- [ ] Test sustained load above the threshold.
- [ ] Test retry storms and failure scenarios.
- [ ] Test long-running requests if concurrency limits are used.
- [ ] Confirm that one caller cannot starve others during stress.

10) Verify observability

- [ ] Emit logs for throttled or rejected requests.
- [ ] Track metrics for allowed, queued, and rejected requests.
- [ ] Include the policy name or route in telemetry.
- [ ] Monitor whether rejections increase during incidents.
- [ ] Set alerts for unexpected throttle rates or rising queue depth.
- [ ] Make sure operators can distinguish healthy enforcement from misconfiguration.

11) Review fairness and side effects

- [ ] Check whether multiple callers can collapse into one partition key.
- [ ] Evaluate whether shared NATs or proxies cause false positives.
- [ ] Confirm that one tenant or user cannot dominate a shared budget.
- [ ] Make sure expensive endpoints have protection proportional to their cost.

- [ ] Validate that legitimate high-volume clients still have a workable path.

12) Prepare for rollout

- [ ] Start with a conservative pilot on selected endpoints or tenants.
- [ ] Roll out incrementally instead of enabling broad enforcement all at once.
- [ ] Communicate the policy to internal consumers and external clients if applicable.
- [ ] Document fallback behavior if throttling is too aggressive.
- [ ] Review logs and metrics after rollout and adjust values as needed.

13) Production readiness checks

- [ ] Can you state what resource is being protected in one sentence?
- [ ] Can you state which requests share the same budget?
- [ ] Can you state what happens when the budget is exceeded?
- [ ] Can clients tell they were throttled and what to do next?
- [ ] Is the policy consistent with authentication, routing, and infrastructure limits?
- [ ] Have you tested realistic traffic, incident behavior, and retry patterns?

Quick decision guide

- Use a throughput-based policy when the question is: ?How many requests can this caller make??
- Use a concurrency policy when the question is: ?How much work can run at once??
- Use a partitioned policy when different callers need separate budgets.
- Prefer identity-based keys when authenticated identity is trustworthy.

- Prefer layered controls when you need both broad protection and precise policy enforcement.

Final go-live check

Before production, confirm all of the following:

- [ ] The policy matches the actual operational problem.
- [ ] The partition key is appropriate and stable.
- [ ] The algorithm matches the traffic pattern.
- [ ] The rejection path is clear and documented.
- [ ] The monitoring signals are in place.
- [ ] The rollout plan includes a way to tune the policy safely.

If any of these are unclear, revise the policy before enabling it broadly.