Fake Developer Meeting Transcript

Dev A: So, about the caching layer—we have two options: 1. Use Redis for external caching. 2. Keep it in-process with a simple dict plus LRU eviction.

Dev B: Right. Redis gives us persistence and easy horizontal scaling, but it adds network latency and a whole new dependency to maintain.

Dev A: True. The in-process cache would be blazing fast—no network roundtrips—but it doesn't scale across instances. Each server would have its own copy, so we might get cache incoherence.

Dev B: Yeah, that's my concern. If we spin up more containers under load, each one will warm up its cache separately. We'll see a lot of repeated DB hits until the caches converge.

Dev A: On the flip side, Redis adds some operational overhead: we need to run it in HA mode, set up monitoring, handle failover, and make sure latency stays consistent.

Dev B: Also, Redis lets us do smarter eviction strategies and time-based expiry. With a dict LRU, we'll have to implement that ourselves.

Dev A: Another factor: cost. Running Redis in production means more infra and potentially higher cloud bills. An in-memory cache is free, but with the tradeoff of limited capacity.

Dev B: Let's quantify. Our hot set of data is about 200 MB. Each app container has about 1 GB free headroom, so an in-process LRU is viable. If we grow beyond that, Redis starts looking like the safer option.

Dev A: Good point. Maybe we can start with in-process caching, since it's dead simple, and put a feature flag in place to swap in Redis later if we need distributed coherence.

Dev B: Agreed. We get the low latency benefits now, and we don't commit to extra infra until we actually need it.

Conclusion

The team decided to implement an in-process LRU cache first, since it is simple, low-latency, and sufficient for the current data size. They will add a feature flag abstraction layer, making it easy to migrate to Redis later if scale demands distributed caching or more advanced eviction policies.