I Created a SOQL Query Builder That Was Immediately Adopted Company Wide

Context

Salesforce sat at the center of our Kotlin codebase. A lot of engineers touched SOQL every sprint.

The problem was not that we could not query Salesforce. The problem was that everybody had their own way of doing it.

We had "query builders," but that label was generous. Some queries used shared helper functions. Some manually stitched together strings. Some mixed both. That meant two engineers solving the same problem could produce two completely different implementations.

Here is what that inconsistency actually looked like.

Old query style #1: helper-driven composition

```kotlin

fun createQuery(relevantIds: List, contactTypes: List?): SoqlQuery {

val caseOrWorkOrderCondition =

    orOp(

        inFilter("CaseId", relevantIds),

        inFilter("WorkOrderId", relevantIds),

    )

val contactCondition =

    subQueryIn(

        filter = contactTypes,

        outerField = "ContactId",

        subQueryTable = "AccountContactLink",

        selectField = "ContactId",

        filterField = "ContactType",

    )

return SoqlQuery(listOfNotNull(BASE\_SELECT, whereClause(caseOrWorkOrderCondition, contactCondition)))

}
```

Old query style #2: manual string assembly

```kotlin
fun createQuery(request: WorkOrderSearchRequest): SoqlQuery {

val clause =

    sequenceOf(

        fromFilter(request.workOrderIds, "Id"),

        fromFilter(request.caseIds, "Case.Id"),

             fromFilterNot(if  (request.onlyOpen  ==  true)  CLOSED\_STATUSES  else  null,
"Status"),

        fromFilter(request.assetIds, "Asset.Name"),

    ).filterNotNull().joinToString(" AND ")

val where =

    if (clause.isBlank()) {

        ""

    } else {

        " WHERE "

    }

return SoqlQuery(BASE\_QUERY + where + clause)

}
```

These two queries were doing the same basic job: build a `SELECT ... FROM ... WHERE ...` statement with optional filters.

But they did it through two different mental models.

- One relied on helper functions like `orOp(...)`, `subQueryIn(...)`, and `whereClause(...)`.
- The other built a list of clause fragments, joined them manually, then handled spacing and ` WHERE` insertion itself.

That was the real problem. A new engineer could not learn one pattern and apply it everywhere. Reviewers could not quickly pattern-match. Every query change required re-reading custom assembly logic.

That is how teams quietly waste time.

Action

I treated this as a staff-level standardization problem, not a syntax cleanup.

The first step was identifying the real need. Not "we should clean this up someday." The real need was that a core integration surface used by multiple teams had no standard way to do a routine task safely. That was costing us delivery speed, review time, and production confidence.

So I audited the existing query creators and the helper layer underneath them. I wanted to know exactly what engineers were doing in the wild: optional filters, grouped boolean logic, nested subqueries, aggregation, ordering, and all the little edge cases that usually get lost in a "simple" abstraction.

Once I had the landscape, I designed one Kotlin DSL that matched the shape of SOQL directly.

Instead of asking engineers to assemble strings, I gave them a way to describe a query.

```kotlin
fun createQuery(relevantIds: List, contactTypes: List?): SoqlQuery =

soql {
    select(FIELDS)
    from("VoiceCall")
    where {
        "CaseId" isIn relevantIds
        or { "WorkOrderId" isIn relevantIds }
    }
    and {
        "ContactId" isIn {
            select("ContactId")
            from("AccountContactLink")
            where { "ContactType" isIn contactTypes }
        }
    }
}
```

And the work-order query that used to hand-assemble strings became this:

```kotlin
fun createQuery(request: WorkOrderSearchRequest): SoqlQuery =

soql {
    select(FIELDS)
    from("WorkOrder")
    where { "Id" isIn request.workOrderIds }

    and { "Case.Id" isIn request.caseIds }

    and { "Status" notIn if (request.onlyOpen == true) CLOSED\_STATUSES else null }

    and { "Asset.Name" isIn request.assetIds }

}
```

\`\`\`

That change did three things at once:

- It made the query readable. You could look at the code and immediately see the SOQL shape.
- It made the common path consistent. Every query creator now used the same grammar: ` select`, `from`, `where`, `and`, `or`, `groupBy`, `orderBy`.
- It moved correctness into the abstraction. Empty filters were skipped consistently. Grouping logic was explicit. Query validation happened at the query object boundary instead of being left to whoever happened to be writing string concatenation that day.

Coming up with the builder was not the hard part. Getting broad adoption was.

This is where partnering mattered. I worked with senior and staff engineers across the teams that owned these queries. The concern was reasonable: "How do we know the new builder will not subtly change behavior in production?"

My answer was simple: tests are the migration strategy.

Before I changed query creators, I locked down query output with tests. Once the outputs were pinned, the migration stopped being a leap of faith. It became a mechanical replacement: same output, better abstraction. That changed the conversation from "this is risky" to "the risk is already bounded."

That buy-in mattered, but alignment by itself does not create adoption. Engineers still have to spend time migrating their code, and that is where a lot of good internal platforms stall out.

So I removed that cost too.

I handled the migration work myself. I used code generation agents to accelerate the repetitive rewrites, then reviewed the generated changes against the pinned query outputs. That let me move the whole codebase without asking every team to stop their roadmap and pay a migration tax for an improvement they already agreed with in principle.

That was the adoption strategy: zero effort on their end.

I did not ask a dozen engineers to come back later and clean up their queries. I brought them a PR where the hard work was already done, the behavior was already validated, and the review surface was clear.

That is why I did not do a gradual rollout. A slow rollout would have preserved the exact thing I was trying to remove: two ways of building SOQL in the same codebase. Once the tests were there and the migration work was handled, the lowest-risk move was to migrate everything in one PR.

That shipped as the query builder merge request: one new builder, one deleted helper layer, and query creators across the Salesforce surface migrated in one pass.

I also left obvious seams in the design instead of trying to predict every future use case. That mattered later. Other engineers extended the builder without me:

- one follow-up added null predicates like `isNull()` and `isNotNull()`
- another added upper-bound date filtering with `lessThanOrEqual(...)`

That was not a side effect. That was the point. I did not want a library only I could safely change.

Result

The migration landed with zero production bugs.

After that:

- SOQL work got faster. The average task dropped from roughly 3 story points to 2.
- Code reviews got faster because reviewers were checking a standard pattern, not reverse-engineering custom string logic.
- Onboarding got easier because there was one way to write queries, not ten.
- Query-related defects dropped because the builder removed whole categories of assembly mistakes.
- The builder kept growing because other engineers could extend it without redesigning it.

The biggest win was not prettier Kotlin.

The biggest win was that I turned a messy, team-by-team problem into a standardized platform capability with near-zero adoption cost for everybody else.

Learning

If the problem shows up across teams, standardization is the leverage. I was not optimizing one query. I was removing repeated decision-making from an entire part of the codebase.

Coming up with the abstraction is only half the job. Staff-level work is not just "design something good." It is also "get the organization to trust it."

Buy-in gets easier when you can make the risk concrete. In this case, that meant proving output equivalence with tests instead of arguing from taste.

Code does not automatically create adoption. Sometimes the fastest way to get adoption is to absorb the migration cost yourself so everyone else gets the improvement for free.

Good abstractions reduce thinking, not just lines of code. The builder worked because engineers no longer had to think about spacing, empty clauses, or composition rules every time they touched SOQL.

Leave extension points obvious. The strongest signal that the design worked was that other engineers extended it later without needing me in the loop.