GRAPHQL CONF 2026

# From Query to Conversation:
# GraphQL as a

**Adam Conrad**
Software Engineering Manager, Meta

**Hugh Nguyen**
Software Engineer, Meta

OVERVIEW

# Agenda


01 The problem

02 The requirements

03 Recursive schema introspection

04 Static subschemas

05 Vector search + dynamic subschemas

06 Conclusion

OUR MISSION

## Give AI agents all of the GraphQL they need so we can interact with our data

01

# How do agents work with GraphQL?

To accomplish everyday work tasks in conversational English

02

# The Requirements

1. **Understand business logic** — what does each enum value mean?
2. **Large SDL** — support a variety of customer needs
3. **Security & privacy** — sensitive user data

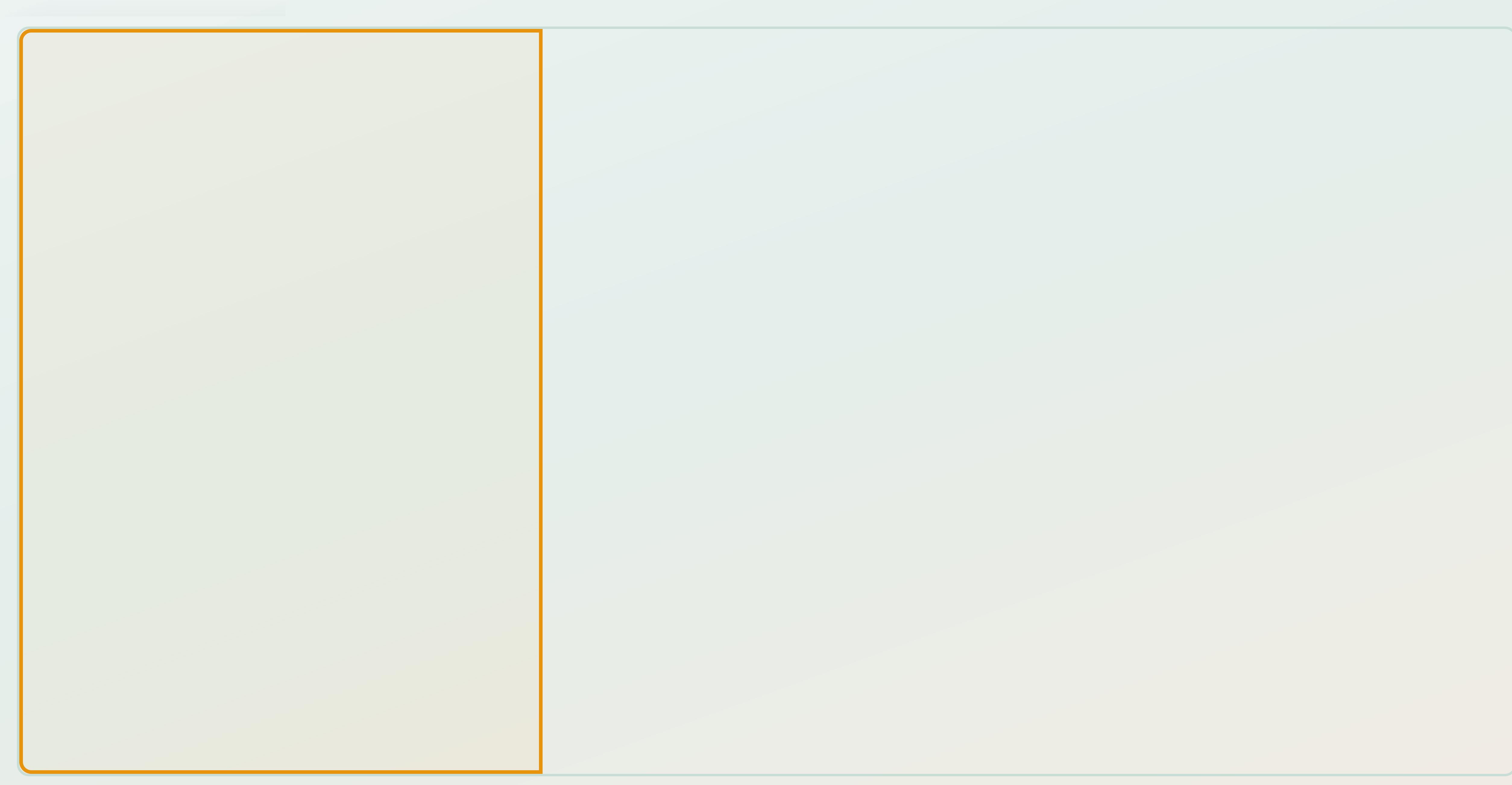
# 03

# Recursive Schema Introspectio

Give the agent tools to explore the schema step by step — like a developer using GraphiQL

03 · RECURSIVE SCHEMA INTROSPECTION

# GraphiQL — Schema Documentation



# GraphQL Schema Introspection

```
type Query {
  __schema: __Schema
  ...
}

type __Schema {
  # MCP tools as GraphQL fields
  search_fields(keyword: String!, parent_type: String!): [__Field!]!
  search_types(keyword: String!): [__Type!]!
}
```

03 · RECURSIVE SCHEMA INTROSPECTION

# “Find my top 5 friends names”

| Agent | MCP Tools |
| --- | --- |

TURN 1

```
→ search_fields( parent_type: "Query", keyword: "me" )
```

```
← me: User
```

TURN 2

```
→ search_fields( parent_type: "User", keyword: "friends" )
```

```
← User.friends( first: Int, orderby: [FriendsOrdering] ): FriendsConnection
```

TURN 3

```
→ search_types( keyword: "FriendsOrdering" )
```

```
← enum FriendsOrdering( "alphabetical", "importance", "recent" )
```

· · ·

TURN 4+

```
→ query { me { friends(first: 5, orderby: [importance]) { nodes { name } } } }
```

**INSIGHT**

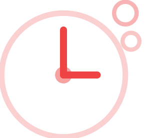
The agent is playing **“GraphQL Pictionary”** — asking for tiny clues one by one. Minimum 4 sequential tool calls just to construct the schema.

# Pros & Cons

### Exhaustive Discovery

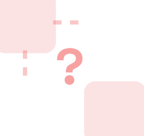
All schema elements are eventually discoverable

**BUT...**



### Slow & Expensive

Requires many turns and tokens




### Lacking Relevancy

Agent lacks knowledge of field utility and required combinations

# 04

# Static Subschemas

Give the agent the relevant SDL up front — instant capabilities

"Find my top 5 friends names"



```graphql
# Schema in SDL format
type Query {
  me: User
}

type User {
  id: ID!
  name: String
  friends(
    first: Int,
    orderby: [FriendsOrdering]
  ): FriendsConnection
}

type FriendsConnection {
  nodes: [User!]!
}

enum FriendsOrdering {
  alphabetical, importance,
recent
}
```

→

**0 TOOL CALLS**

```graphql
# Generated query
query TopFriendsQuery {
  me {
    friends(first: 5, orderby: [importance])
{
      nodes { name }
    }
  }
}
```

KEY INSIGHT

Show the LLM a SDL, and it instantly gains the a[illegible] to query that domain.

# More schema = more capabilities

## Add mutation schema...

```
type Mutation {
  createPost(input: CreatePostInput!):
Post
}

input CreatePostInput {
  body: String!
  privacy: PostPrivacy
}

enum PostPrivacy {
  PUBLIC, FRIENDS, ONLY_ME
}
```

## ...agent can now mutate!

```
mutation CreateFriendsOnlyPost {
  createPost(input: {
    body: "Hello from GraphQL!"
    privacy: FRIENDS
  }) {
    id
    body
    privacy
  }
}
```

IMPLICATION

## Every SDL you add unlocks a new capa[illegible].

# But not all of them are needed at the same time

Context Window 100%

| GraphQL Schema | remaining for thinking, documents, memory... |
| --- | --- |

- Searching for friends? You don't need the Post mutation schema
- We need a way to **load schemas on demand**

# Carve the schema into focused pieces

## Subschemas (loaded on demand)

② Calls load_subschema("USER") — types returned

### USER

```
User { id, name, friends(…) }
FriendsConnection { nodes… }
enum FriendsOrdering { … }
```

### EVENTS

```
Event { id, title, start_time… }
EventAttendee { user, status }
enum RSVPStatus { … }
```

### POSTS

```
Post { id, body, author… }
input CreatePostInput { body, privacy }
enum PostPrivacy { … }
```

## Entrypoint (always in system prompt)

```
type Query {
  me: User @subschema(USER)

  event(id: ID!): Event
@subschema(EVENTS)
  load_subschema(name: String!): String
}

type Mutation {
  createPost(…): Post @subschema(POSTS)
  rsvpEvent(…): Event @subschema(EVENTS)
}
```

① Agent reads directive

## WHAT WE GAINED

### Fast

1 tool call instead of 4

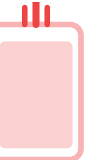
### Progressive

Only load what the task needs

## BUT...

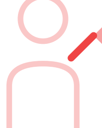
### Maintenance Nightmare

Field renamed in code → SDL silently breaks



### Token Hungry

Fixed token cost in system prompt



### Not Exhaustive

New capabilities need manual subschemas

# 05

# Vector Search + Dynamic Subschemas

Let the agent discover and generate the schema it needs — on the fly

# The agent searches for what it needs

```
search_schema(query: "create event")
```
TOOL CALL

Vector similarity search across the entire schema

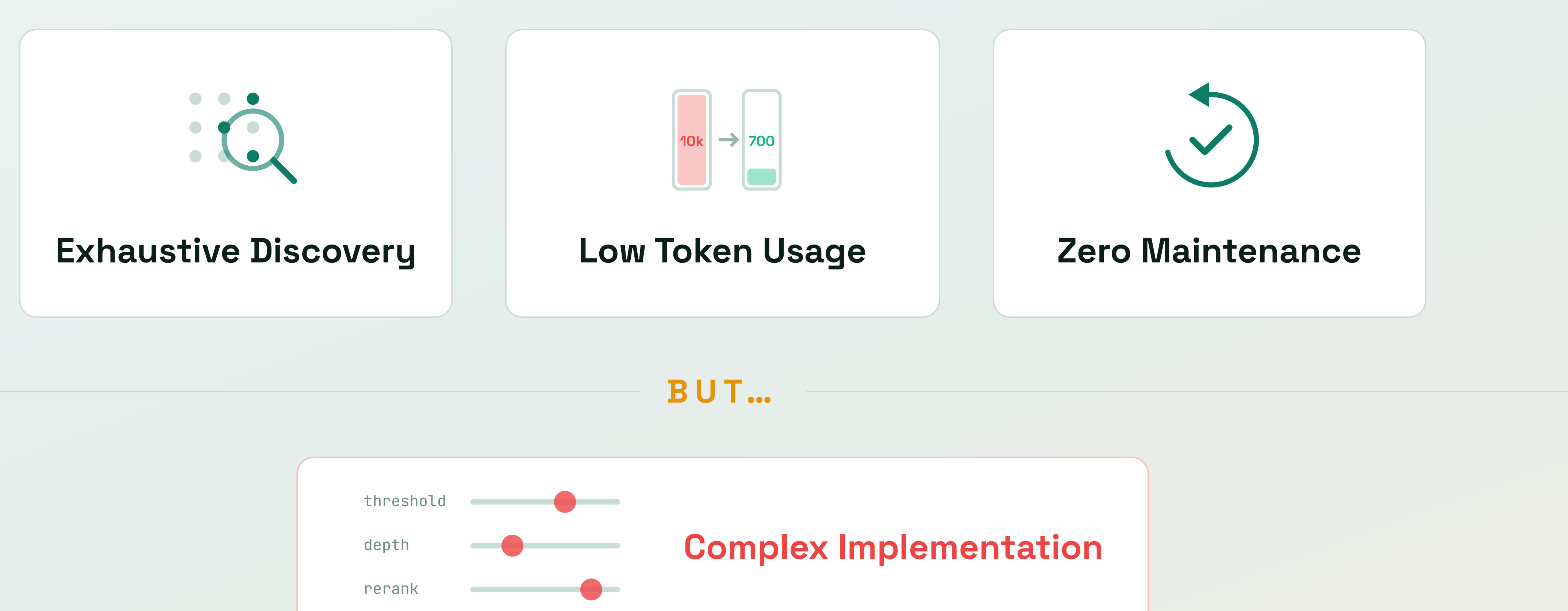
RESULTS

```
type Mutation {
  createEvent(…): Event # score: 0.94
  rsvpEvent(…): Event # score: 0.81 — semantic match
}
```

Natural language in, relevant fields out — from the *entire* schema.

## Before & after: static vs. dynamic

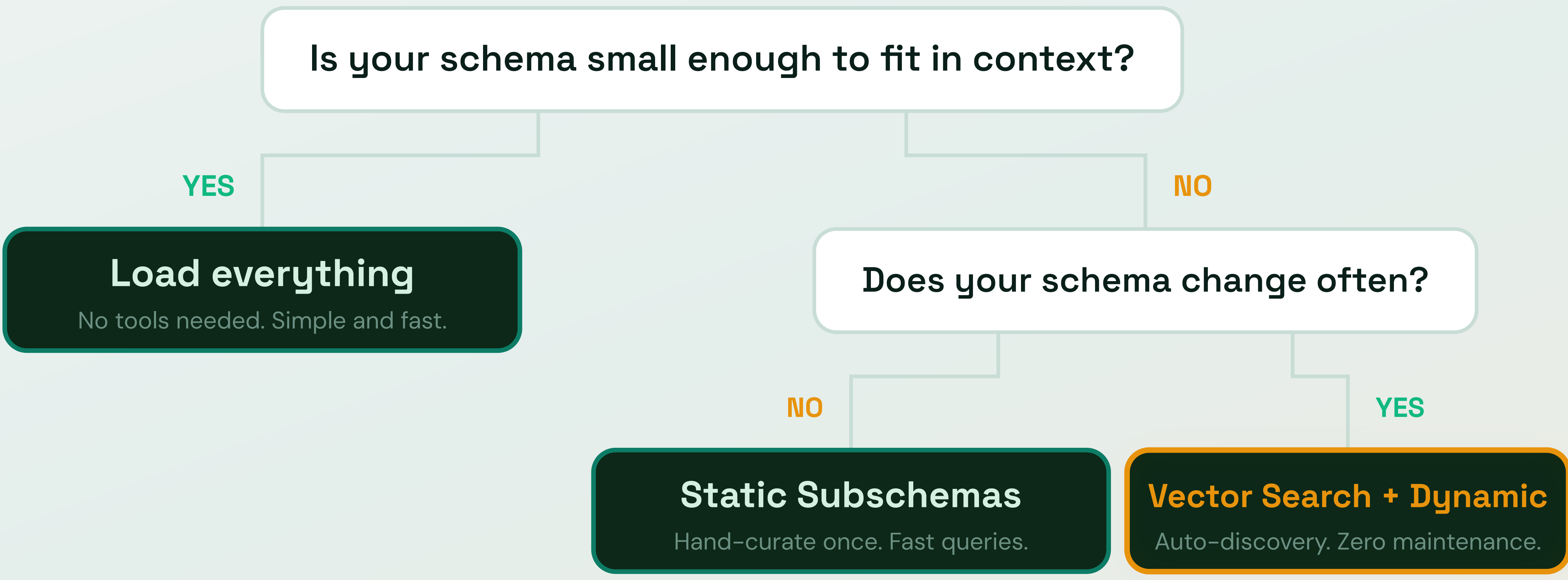
| STATIC SUBSCHEMA | VECTOR SEARCH + DYNAMIC |
| --- | --- |
| Entrypoint file (always in system prompt) ~10,000 tokens | `search_schema(query: "create event")` ~700 tokens |
| `load_subschema("EVENTS")` | `generate_subschema("createEvent")` |


WHAT WE GAINED
10k
700
Exhaustive Discovery
Low Token Usage
Zero Maintenance
BUT...
threshold
depth
rerank
Complex Implementation

## What should you use?

| | INTROSPECTION | STATIC SUBSCHEMAS | VECTOR SEARCH + DYNAMIC |
|---|---|---|---|
| Exhaustive | ✓ | ✗ | ✓ |
| Speed | Slow | ✓ | ✓ |
| Token cost | High | Medium | Low |
| Maintenance | None | High | None |
| Simplicity | Medium | Medium | Complex |
| Scalable | ✓ | Somewhat | ✓ |

# How to choose



Adam Conrad & Hugh Nguyen

> “These are the best ideas we have… **for now!**”