Build Your Own GraphQL Engine: Design Document

Overview

This system builds a complete GraphQL execution engine that can parse GraphQL queries, reflect database schemas automatically, and compile queries to efficient SQL. The key architectural challenge is bridging the conceptual gap between GraphQL's hierarchical query model and relational databases' tabular structure while maintaining performance and type safety.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** This section provides foundational context for all milestones, establishing the problem domain and mental models that will guide the entire project.

GraphQL has revolutionized API development by providing a declarative, client-driven query language that enables applications to request exactly the data they need. However, building a robust GraphQL execution engine—particularly one that efficiently bridges GraphQL queries to relational databases—is a complex undertaking requiring expertise across parsing, type systems, query execution, database introspection, and query compilation. This project addresses the educational gap between using GraphQL libraries and understanding their internals by building a complete, simplified GraphQL engine from the ground up.

Mental Model: Restaurant Menu vs. Kitchen

Imagine a restaurant with two distinct spaces: the **dining room** where customers order from a menu, and the **kitchen** where chefs prepare meals. This analogy perfectly captures the GraphQL engine architecture:

- **GraphQL Query = Customer Order**: The customer (client) selects specific items from the menu (GraphQL schema) in a particular combination. They don't specify *how* the food should be prepared—just what they want in the final result. A GraphQL query similarly declares what data is needed without specifying how to fetch it from the database.
- **GraphQL Schema = Restaurant Menu**: The menu defines all possible dishes (types) and their available customizations (fields with arguments). Some dishes require special preparation (resolvers), while others are simple combinations of existing items. The menu's structure determines what combinations are valid (type validation).
- **GraphQL Engine = Kitchen System**: The kitchen receives the order and must:
  1. **Parse the order** (Parser) - Understand exactly which dishes and modifications were requested
  2. **Check inventory** (Type System) - Verify all ingredients (data types) are available and compatible
  3. **Plan preparation** (Execution Engine) - Determine the optimal sequence of cooking steps (resolvers)
  4. **Gather ingredients** (Schema Reflection) - Auto-discover what's in the pantry (database schema)
  5. **Cook efficiently** (SQL Compilation) - Combine preparation steps to avoid redundant work (N+1 problem)

> **Key Insight:** Just as a skilled kitchen can prepare multiple parts of an order in parallel (appetizers while entrees cook) and combine similar preparation steps (chopping all vegetables at once), a good GraphQL engine executes independent fields concurrently and batches related database operations.

The GraphQL-Database Translation Problem

The fundamental challenge in building a GraphQL engine lies in the **impedance mismatch** between GraphQL's hierarchical, graph-based query model and relational databases' tabular, set-based structure. This mismatch creates several specific technical challenges:

Challenge	GraphQL Perspective	Relational Database Perspective	Technical Consequence
Data Shape	Nested objects with arbitrary depth	Flat tables with foreign key relationships	Requires JOINs or multiple queries to reconstruct hierarchy
Field Selection	Client selects exact fields needed	SQL `SELECT *` fetches all columns	Must generate SQL with only requested columns to avoid over-fetching
Nullability	Explicit non-null (`!`) types with propagation rules	NULL-able columns with three-valued logic	Must enforce GraphQL's strict null propagation, which differs from SQL semantics
Type System	Strong, static types with interfaces and unions	Primitive types with constraints and referential integrity	Requires mapping SQL types to GraphQL scalars and enforcing type safety
Query Structure	Recursive selection sets with fragments	Linear SQL statements with fixed structure	Must flatten nested selections into SQL JOIN hierarchy

Concrete Example: The N+1 Query Problem

Consider a simple GraphQL query fetching blog posts with their authors:

```graphql
query {
  posts {
    title
    content
    author {
      name
      email
    }
  }
}
```

A naive implementation might execute:

1. `SELECT id, title, content, author_id FROM posts` (1 query)
2. For each post: `SELECT name, email FROM authors WHERE id = ?` (N queries)

This results in **N+1 queries** (1 for posts, N for authors). The engine must instead generate:

```sql
SELECT posts.title, posts.content, authors.name, authors.email

FROM posts

LEFT JOIN authors ON posts.author_id = authors.id
```

But this becomes more complex with one-to-many relationships (comments on posts) requiring lateral joins or separate batched queries. The translation problem involves:

- Analyzing the GraphQL AST to understand relationship patterns
- Generating optimal SQL JOIN structures
- Batching independent lookups using IN clauses
- Maintaining type safety and nullability semantics

Architecture Decision: Translation Strategy

> **Decision: AST-to-SQL Direct Compilation**

- **Context**: We need to translate GraphQL queries to database queries with minimal overhead. Three main approaches exist: resolver-per-field (naive N+1), batched resolvers with DataLoader, or direct AST-to-SQL compilation.
- **Options Considered**:
  1. **Resolver-per-field**: Each field has a resolver function; simple but leads to N+1 queries
  2. **DataLoader pattern**: Resolvers request data, DataLoader batches similar requests; good for caching but adds abstraction layer
  3. **Direct AST-to-SQL compilation**: Analyze the entire query, generate optimized SQL; complex but maximum performance

- **Decision**: Direct AST-to-SQL compilation for the core engine, with optional DataLoader for custom resolvers
- **Rationale**:
  - Educational value: Compilation demonstrates query optimization principles clearly
  - Performance: Single optimized SQL query minimizes round-trips
  - Predictability: Deterministic SQL generation easier to debug than distributed resolver timing
- **Consequences**:
  - Must implement full query analysis and SQL generation
  - Less flexible for custom business logic in resolvers
  - Better for database-heavy applications, worse for microservice aggregation

Option	Pros	Cons	Why Chosen?
Resolver-per-field	Simple to implement, flexible for custom logic	Severe N+1 problem, poor performance	Rejected for core engine (educational anti-pattern)
DataLoader pattern	Good for microservices, handles caching	Added complexity, still multiple queries	Supplemental pattern for custom resolvers
Direct compilation	Maximum performance, single query, deterministic	Complex implementation, less flexible	**Primary approach** - best for learning and performance

Existing Solutions Comparison

Several production GraphQL engines address the database translation problem with different architectures and trade-offs. Understanding these helps clarify our design choices and educational objectives.

Hasura: Declarative Configuration Engine

Hasura takes a **declarative, metadata-driven approach**:

- **Architecture**: PostgreSQL-focused, generates GraphQL schema from database metadata with fine-grained permissions
- **Translation Strategy**: AST-to-SQL compilation with extensive optimizations (predicate pushdown, join elimination)
- **Strengths**:
  - Excellent performance through query compilation
  - Real-time subscriptions via PostgreSQL LISTEN/NOTIFY
  - Rich permission system declaratively configured
- **Limitations**:
  - Primarily PostgreSQL-only (though expanding)
  - Limited custom business logic without external services
  - Opinionated schema generation

```sql
-- Hasura internally generates SQL like this from GraphQL:
SELECT json_agg(__root) FROM (
  SELECT json_build_object(
    'id', posts.id,
    'title', posts.title,
    'author', author_obj
  ) AS __root
  FROM posts
  LEFT JOIN LATERAL (
    SELECT json_build_object('name', authors.name) AS author_obj
    FROM authors WHERE authors.id = posts.author_id
  ) ON true
) AS __root
```

PostGraphile: Reflection-Driven with Plugins

PostGraphile uses **PostgreSQL reflection and convention-over-configuration**:

- **Architecture**: Reflects database schema, uses PostgreSQL comments for metadata, plugin system for extensibility
- **Translation Strategy**: Smart defaults with plugin-based query building
- **Strengths**:
  - Extremely fast schema generation from existing databases
  - Customizable via plugin ecosystem
  - Leverages PostgreSQL features (ROW SECURITY, etc.)
- **Limitations**:
  - PostgreSQL-specific
  - Learning curve for customization
  - Less control over exact GraphQL schema shape

Custom Implementation (Apollo/GraphQL-JS)

The **resolver-based approach** used by most custom GraphQL servers:

- **Architecture**: Manual schema definition, resolver functions for each field
- **Translation Strategy**: Manual ORM/query calls within resolvers, DataLoader for batching
- **Strengths**:
  - Complete control over business logic
  - Database-agnostic (works with any data source)
  - Mature ecosystem (Apollo, GraphQL-JS)
- **Limitations**:
  - Manual N+1 optimization required
  - Boilerplate code for common patterns
  - Performance depends on developer discipline

Our Educational Approach

Our engine borrows concepts from all three but prioritizes **educational transparency**:

Aspect	Hasura (Production)	PostGraphile (Production)	Custom (Typical)	Our Engine (Educational)
Schema Generation	Metadata-driven	Reflection + conventions	Manual definition	**Reflection + explicit mapping**
Query Translation	AST-to-SQL compiler	Plugin-based compilation	Resolver functions	**AST-to-SQL with visible steps**
Performance Focus	Maximum optimization	Good defaults with plugins	Developer-dependent	**Optimized but transparent**
Learning Value	Black box	Convention magic	Business logic focus	**White box with clear algorithms**
Database Support	PostgreSQL-first	PostgreSQL-only	Any	**Adapter pattern with reference impl**
Custom Logic	Remote schemas/actions	Plugin system	First-class	**Hybrid: compiled + resolver fallback**

> **Design Philosophy**: Our engine makes compilation steps explicit—learners can trace a GraphQL query through parsing, type checking, AST analysis, JOIN generation, and SQL parameterization. Where Hasura hides complexity for production simplicity, we expose it for learning.

Why Build From Scratch?

Existing solutions work well for production but obscure the underlying mechanics:

1. **Abstraction Layers Hide Fundamentals**: Tools like Hasura generate SQL automatically, but developers don't learn *how* the translation works
2. **Missing Mental Models**: Without understanding the compilation pipeline, debugging performance issues becomes guesswork
3. **Educational Gap**: Tutorials focus on using GraphQL, not implementing execution engines
4. **Customization Limitations**: When existing tools don't fit a use case, developers lack the foundation to build their own solutions

This project fills that gap by building a **pedagogical reference implementation** that:

- Shows each transformation step explicitly
- Provides extensibility points learners can modify
- Includes common pitfalls and debugging guidance
- Balances simplicity with real-world concerns

Implementation Guidance

> **Note**: This section provides initial setup and mental model implementation. The detailed component implementations will follow in subsequent sections.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option	Why for This Project
Language	Python 3.9+	Rust or Go	Python chosen for readability and educational accessibility
HTTP Server	Flask (synchronous)	FastAPI (async)	Flask's simplicity reduces cognitive load for learning core concepts
Database	SQLite (file-based)	PostgreSQL (production)	SQLite requires no setup, focuses on SQL generation not connection management
SQL Building	String concatenation	SQLAlchemy Core	Starting with strings shows the raw SQL; can evolve to builder pattern
Testing	unittest/pytest	pytest with async	unittest is standard library, sufficient for learning
Type Hints	Basic type hints	mypy strict	Type hints improve readability without complexity

B. Recommended File/Module Structure

Create this directory structure from the beginning to maintain separation of concerns:

```
graphql-engine/
├── README.md
├── pyproject.toml           # Python project configuration
├── requirements.txt         # Dependencies
├── src/
│   ├── graphql_engine/
│   │   ├── __init__.py
│   │   ├── main.py          # HTTP server entry point
│   │   ├── parser/          # Milestone 1
│   │   │   ├── __init__.py
│   │   │   ├── lexer.py     # Tokenizer
│   │   │   ├── parser.py    # Recursive descent parser
│   │   │   └── ast.py       # AST node definitions
│   │   ├── type_system/     # Milestone 2
│   │   │   ├── __init__.py
│   │   │   ├── types.py     # GraphQL type classes
│   │   │   ├── schema.py    # Schema construction/validation
│   │   │   └── introspection.py
│   │   ├── execution/       # Milestone 3
│   │   │   ├── __init__.py
│   │   │   ├── executor.py  # Query execution engine
│   │   │   ├── resolvers.py # Default resolver logic
│   │   │   └── dataloader.py# DataLoader implementation
│   │   ├── reflection/      # Milestone 4
│   │   │   ├── __init__.py
│   │   │   ├── reflector.py # Database schema reflection
│   │   │   ├── sql_types.py # SQL-to-GraphQL type mapping
│   │   │   └── relationships.py
│   │   ├── compilation/     # Milestone 5
│   │   │   ├── __init__.py
│   │   │   ├── compiler.py  # AST-to-SQL compiler
│   │   │   ├── sql_builder.py
│   │   │   └── query_plan.py
│   │   └── utils/
│   │       ├── __init__.py
│   │       ├── errors.py    # Custom exception classes
│   │       └── logging.py   # Debug logging helpers
└── tests/
    ├── __init__.py
    ├── test_parser.py
    ├── test_type_system.py
    ├── test_execution.py
    ├── test_reflection.py
    └── test_compilation.py
```

C. Infrastructure Starter Code

Create a simple HTTP server that will accept GraphQL queries. This isn't the learning focus but is necessary to test the engine:

PYTHON

```python
# src/graphql_engine/main.py

"""
Simple HTTP server for the GraphQL engine.
This provides a test harness - the real learning is in the components.
"""
from flask import Flask, request, jsonify
import json
from typing import Dict, Any

app = Flask(__name__)

# These will be implemented in later milestones
from graphql_engine.parser import parse_query
from graphql_engine.type_system import GraphQLSchema
from graphql_engine.execution import execute_query

# In-memory schema cache (will be populated via reflection)
schema_cache: Dict[str, GraphQLSchema] = {}

@app.route('/graphql', methods=['POST'])
def graphql_endpoint():
    """
    Standard GraphQL HTTP endpoint.
    Expects JSON with 'query' and optionally 'variables', 'operationName'.
    """
    try:
        data = request.get_json()

        if not data or 'query' not in data:
            return jsonify({
                'errors': [{'message': 'Must provide query parameter'}]
            }), 400

        query = data['query']
        variables = data.get('variables') or {}
        operation_name = data.get('operationName')

        # TODO: In Milestone 4, this will reflect the database
```

```
        # For now, use a placeholder schema
        if 'default' not in schema_cache:
            schema_cache['default'] = GraphQLSchema()  # Will be populated later

        schema = schema_cache['default']

        # Parse the query (Milestone 1)
        document_ast = parse_query(query)

        # Execute the query (Milestone 3)
        result = execute_query(
            schema=schema,
            document_ast=document_ast,
            variable_values=variables,
            operation_name=operation_name
        )

        return jsonify(result)

    except Exception as e:
        # TODO: Better error formatting based on error type
        app.logger.exception("GraphQL execution error")
        return jsonify({
            'errors': [{'message': str(e)}]
        }), 500

@app.route('/health')
def health_check():
    """Simple health endpoint for deployment."""
    return jsonify({'status': 'healthy'})

if __name__ == '__main__':
    app.run(debug=True, port=5000)
```

D. Core Logic Skeleton Code

Here's the initial AST structure that will be used throughout the engine. Learners will expand this in Milestone 1:

PYTHON

```python
# src/graphql_engine/parser/ast.py

"""
Abstract Syntax Tree node definitions for GraphQL.
These data structures represent parsed GraphQL queries.
"""

from dataclasses import dataclass, field
from typing import List, Optional, Dict, Any, Union

@dataclass
class Location:
    """Source location for error reporting."""
    line: int
    column: int

    def __str__(self) -> str:
        return f"line {self.line}, column {self.column}"

@dataclass
class Node:
    """Base class for all AST nodes."""
    loc: Optional[Location] = None  # Source location (optional in some cases)

@dataclass
class Document(Node):
    """Root node representing a complete GraphQL document."""
    definitions: List['Definition'] = field(default_factory=list)

@dataclass
class Definition(Node):
    """Base for definitions (operations, fragments)."""
    pass

@dataclass
class OperationDefinition(Definition):
    """An operation (query, mutation, subscription)."""
    operation_type: str  # 'query', 'mutation', 'subscription'
    name: Optional[str] = None
    variable_definitions: List['VariableDefinition'] = field(default_factory=list)
```

```
    directives: List['Directive'] = field(default_factory=list)
    selection_set: 'SelectionSet' = None  # Will be set during parsing

@dataclass
class SelectionSet(Node):
    """A set of selections (fields, fragment spreads)."""
    selections: List['Selection'] = field(default_factory=list)

@dataclass
class Selection(Node):
    """Base for selections within a selection set."""
    pass

@dataclass
class Field(Selection):
    """A field selection within a selection set."""
    name: str = ""
    alias: Optional[str] = None
    arguments: List['Argument'] = field(default_factory=list)
    directives: List['Directive'] = field(default_factory=list)
    selection_set: Optional[SelectionSet] = None  # None for scalar fields

# TODO: Add remaining AST node classes:
# - FragmentDefinition
# - FragmentSpread
# - InlineFragment
# - VariableDefinition
# - Argument
# - Directive
# - Value nodes (IntValue, StringValue, Variable, etc.)
```

E. Language-Specific Hints

1. **Python Type Hints**: Use `from typing import ...` extensively. This improves readability and helps catch errors early.
2. **Dataclasses**: Use `@dataclass` for AST nodes and data structures—they auto-generate `__init__`, `__repr__`, and support default values.
3. **Error Handling**: Create a custom exception hierarchy in `utils/errors.py`:

```python
class GraphQLError(Exception):
    """Base class for all GraphQL errors."""
    pass


class GraphQLSyntaxError(GraphQLError):
    """Raised when query syntax is invalid."""
    pass


class GraphQLValidationError(GraphQLError):
    """Raised when query fails validation against schema."""
    pass
```

4. **Testing**: Write tests alongside implementation. Use `unittest.TestCase` with descriptive method names.

F. Milestone Checkpoint

After setting up the project structure and starter code:

1. **Verify Setup**:

```bash
cd graphql-engine
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install flask
python src/graphql_engine/main.py
```

2. **Test HTTP Server**:

```bash
curl -X POST http://localhost:5000/health
# Should return: {"status": "healthy"}


curl -X POST http://localhost:5000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ __typename }"}'
# Will fail with "NotImplementedError" - this is expected at this stage
```

3. **Expected Behavior**: The server starts without errors. The `/graphql` endpoint returns a 500 error with "NotImplementedError" since the parser and executor aren't built yet. This confirms the infrastructure works.

4. **Common Setup Issues**:
   - **Import errors**: Ensure `src/graphql_engine/__init__.py` exists (can be empty)
   - **Module not found**: Run from project root or set `PYTHONPATH`
   - **Port already in use**: Change port in `app.run(port=5001)`

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"ModuleNotFoundError: No module named 'graphql_engine'"	Python can't find the module	Check `sys.path`, ensure running from project root	Run with `PYTHONPATH=src python src/graphql_engine/main.py`
Flask server starts but crashes on first request	Missing dependencies	Check terminal for import errors	`pip install flask`
AST node attributes are `None` when they shouldn't be	Parser not setting fields correctly	Add debug prints in parser to trace node creation	Ensure each parsing method returns fully populated nodes
SQL query has syntax errors	String concatenation issues	Print generated SQL before execution	Use parameterized queries, not string interpolation

---

Goals and Non-Goals

> **Milestone(s):** This section establishes the foundational scope for all five milestones, defining the absolute minimum required for the system to be considered complete, the quality attributes that guide design decisions, and explicit boundaries that prevent scope creep.

This section defines the precise boundaries of our GraphQL engine—what it must achieve to be functionally complete and what it deliberately excludes to maintain focus on the core educational objectives. Building a GraphQL engine is a complex endeavor with many potential rabbit holes; clear scope definition is essential to deliver a working system while providing deep learning in the fundamental challenges of query languages, type systems, and database integration.

Must-Have Requirements

> **Mental Model:** The **Minimum Viable Kitchen** — Imagine building a restaurant kitchen from scratch. You must have at minimum: a heat source (stove), refrigeration, prep surfaces, and basic utensils. Without any one of these, you cannot function. Our must-have requirements are these irreducible components—the absolute essentials needed to transform a GraphQL query into database results.

The system must implement the complete pipeline from GraphQL query string to database results, corresponding directly to our five milestones. Each milestone delivers a critical, non-negotiable component; together they form a functioning engine.

Requirement Category	Specific Capabilities	Corresponding Milestone	Success Metric
Query Parsing	- Lexical analysis of GraphQL query strings into tokens - Recursive descent parsing producing a complete AST - Support for queries, mutations, fragments, variables, and directives - Accurate source location tracking for error messages	Milestone 1: GraphQL Parser	Given any valid GraphQL query per the specification, the parser produces a correctly structured AST. For invalid queries, it produces descriptive errors with line/column positions.
Type System & Schema	- Representation of all GraphQL type kinds (Object, Scalar, Interface, Union, Enum, InputObject, List, NonNull) - Schema construction with validation for consistency (no circular references, valid interface implementations) - Built-in introspection support (`__schema` , `__type` , `__typename`) - Custom scalar type support with serialization/parsing functions	Milestone 2: Schema & Type System	A GraphQL schema can be defined programmatically, validated without internal contradictions, and correctly respond to introspection queries.
Query Execution	- Field resolver invocation with correct argument coercion and context propagation - Recursive resolution of nested selection sets - Proper null propagation according to GraphQL specification (null bubbles to nearest nullable parent) - Error collection allowing partial results - Concurrent execution of independent sibling fields	Milestone 3: Query Execution Engine	Given a schema with resolver functions, any valid query returns correctly shaped data, handles nulls appropriately, and reports field errors without crashing.
Database Schema Reflection	- Connection to PostgreSQL (or similar SQL database) and introspection of tables, columns, and foreign keys - Automatic mapping of SQL types to GraphQL scalar types (INT→GraphQLInt, VARCHAR→GraphQLString, etc.) - Generation of GraphQL object types from tables with fields for columns and relationships - Creation of primary key query fields and filter arguments for list fields	Milestone 4: Database Schema Reflection	Connecting to a sample database (e.g., a blog with Users, Posts, Comments) produces a complete GraphQL schema with proper types and relationships without manual configuration.
Query-to-SQL Compilation	- Translation of GraphQL field selections to SQL SELECT columns (no over-fetching) - Generation of appropriate SQL JOINs for nested object fields based on foreign keys - Prevention of N+1 queries via batching or single-query JOINs - Parameterized WHERE clauses from GraphQL filter arguments - Support for pagination (LIMIT/OFFSET) and ordering	Milestone 5: Query to SQL Compilation	A GraphQL query requesting nested relationships compiles to a single SQL query (or minimal set) that executes efficiently without Cartesian products or N+1 issues.

Core End-to-End Flow: The system must successfully chain these components: Parse a GraphQL query string → validate it against a schema (either programmatically defined or reflected from a database) → compile it to optimized SQL → execute the SQL → transform database results into the GraphQL response shape. This pipeline represents the complete **impedance mismatch** bridge between GraphQL's hierarchical queries and relational database tables.

Quality Attributes

> **Mental Model:** The **Kitchen's Performance Characteristics** — Beyond having basic equipment, a great kitchen must be: *fast* (low latency from order to plate), *accurate* (orders exactly match what was requested), *adaptable* (can handle special dietary requirements), and *educational* (apprentices can learn why techniques work). These are our quality attributes—the "ilities" that guide how we build, not just what we build.

While functional requirements define *what* the system does, quality attributes define *how well* it does them and influence architectural decisions. For this educational project, we prioritize in this order:

Attribute	Priority	Definition & Measurement	Architectural Implications
Educational Value	Highest	The system's design must expose core concepts clearly, avoid excessive abstraction, and provide learning opportunities at each milestone. Measured by how easily a learner can trace through the codebase and understand each transformation.	- Clear separation of concerns with well-defined component boundaries - Minimal use of "magic" or hidden automation - Extensive comments explaining *why* not just *what* - Straightforward data structures over optimized but obscure ones
Correctness	High	The system must adhere to the GraphQL specification for parsing, validation, and execution semantics. Measured by passing a curated suite of specification-compliance tests.	- Reference implementation of GraphQL spec algorithms (e.g., null propagation, type coercion) - Comprehensive validation at each stage (parse, validate, execute) - Use of property-based testing to ensure edge cases handled
Extensibility	Medium	The architecture must allow for future enhancements (custom directives, additional database adapters, query caching) without major rewrites. Measured by how cleanly new features can be added with minimal modification to existing code.	- Plugin interfaces for database adapters, custom scalar parsers, and directive handlers - Abstract base classes for type definitions and execution strategies - Configuration objects for reflection and compilation options
Performance	Medium-Low	The system should avoid obvious performance pitfalls (N+1 queries, Cartesian products) but need not be production-optimized. Measured by basic benchmarks showing linear scaling with query complexity.	- Query compilation to efficient SQL as primary performance strategy - Simple DataLoader pattern for batching (but not advanced connection pooling) - Acceptable to trade some performance for clarity in educational code

Architecture Decision: Quality Attribute Prioritization

- **Context:** Building a GraphQL engine involves trade-offs between performance, correctness, maintainability, and educational value. Different production systems (Hasura, PostGraphile) prioritize performance and scalability.
- **Options Considered:**
  1. **Production-grade optimization:** Prioritize performance and scalability above all, using advanced compilation techniques, connection pooling, and caching.
  2. **Educational clarity first:** Prioritize readable, straightforward code that clearly demonstrates concepts, accepting performance trade-offs.
  3. **Balanced approach:** Attempt to achieve both good performance and good educational value through careful abstraction.
- **Decision:** Prioritize educational clarity first, with correctness as a close second.
- **Rationale:** This project's primary goal is learning, not deployment to production. Learners must be able to read and understand every component. Over-optimization often obscures fundamental algorithms with complex abstractions. However, correctness is non-negotiable for understanding GraphQL semantics properly.
- **Consequences:** The code will be more verbose and less performant than production systems, but each algorithm will be explicit and traceable. Performance pitfalls will be addressed pedagogically (e.g., demonstrating N+1 problem then fixing it) rather than preemptively hidden.

Option	Pros	Cons	Chosen?
Production-grade optimization	- High performance - Demonstrates real-world scaling techniques	- Complex code obscures fundamentals - Requires advanced database/compiler knowledge	❌
Educational clarity first	- Maximum learning value - Easier debugging for learners - Clear mapping to specification	- Suboptimal performance - May require refactoring for extensions	✅
Balanced approach	- Good trade-off for both goals - More realistic architecture	- Harder to achieve both well - Risk of being neither clear nor fast	❌

Explicit Non-Goals

Mental Model: The **"Not On the Menu" List** — Every restaurant decides what it won't serve: perhaps no vegan options, no late-night service, no catering. These exclusions let the kitchen focus on perfecting its core offerings. Our non-goals are deliberate exclusions that keep the project scope manageable and focused on core educational objectives.

The following features are explicitly **out of scope** for the initial implementation. This prevents scope creep and ensures we deliver a complete, functional engine that teaches the fundamentals without being overwhelmed by advanced features.

Non-Goal	Reason for Exclusion	Potential Future Extension
Authorization & Permissions	Implementing row-level security, role-based access control, or field-level permissions adds significant complexity that distracts from core execution engine concepts.	Could be added as a middleware layer that filters ASTs or wraps resolvers, or via schema directives.
Subscriptions (Real-time)	GraphQL subscriptions require WebSocket handling, pub/sub infrastructure, and stateful connections—a substantial separate system from query execution.	Could be implemented as a separate transport layer using the same parsing/validation but with event-driven resolvers.
Federation & Schema Stitching	Combining multiple GraphQL schemas involves complex query planning and delegation logic that builds upon (rather than teaches) core execution.	Once the base engine is built, federation could be added as a separate query planning layer.
Advanced SQL Features	Window functions, Common Table Expressions (CTEs), stored procedure calls, or database-specific optimizations like materialized views.	These could be exposed via custom directives or special field types after the core compilation works.
Mutation Execution Beyond CRUD	Complex transactional logic, conditional updates, or operations spanning multiple tables in a single mutation.	The basic CRUD mutation generation from reflected schemas provides a foundation; custom resolvers can handle complex logic.
Query Caching	Persistent query caching, response caching with invalidation, or database query result caching.	Simple in-memory caching could be added at the DataLoader or SQL compilation level.
Multiple Database Dialects	Full support for SQLite, MySQL, SQL Server, etc., with their type system differences and SQL variations.	The architecture includes adapter interfaces; we'll focus on PostgreSQL as the reference implementation.
GraphQL Schema Directives	Custom schema directives for code generation, validation, or metadata (e.g., `@deprecated` is built-in, but not custom ones).	Directive visitors could be added to the schema building and query compilation phases.
File Uploads	Multipart form handling or streaming file uploads via GraphQL mutations.	This is typically handled outside GraphQL execution or via custom scalar types.
Performance Monitoring & Tracing	Query complexity analysis, Apollo Tracing, or OpenTelemetry integration.	Could be added as execution context extensions that collect timing data.

> **Key Insight:** The most important non-goal is **production readiness**. This engine is designed for learning, not for deploying to production environments. We deliberately choose simpler implementations that reveal the underlying mechanisms over optimized, black-box solutions. This means our engine will be slower, handle fewer concurrent requests, and lack many features needed for production use—but it will teach you exactly how GraphQL engines work at their core.

Boundary Enforcement: Each component design will reference these non-goals to ensure we don't inadvertently expand scope. For example, the SQL compiler will generate straightforward JOINs rather than attempting to optimize for specific database index structures. The execution engine will use simple concurrent futures rather than sophisticated work-stealing thread pools.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option (Educational Focus)	Advanced Option (If Extending)
Language & Runtime	**Python 3.10+** (standard library, clear syntax, rapid iteration)	Go or Rust (performance, concurrency model)
HTTP Server (for testing)	`http.server` (built-in) or `Flask` (minimal dependencies)	`FastAPI` (async, automatic OpenAPI) or `aiohttp`
Database Adapter	`psycopg2` (PostgreSQL, synchronous, straightforward)	`asyncpg` (async, faster) or SQLAlchemy (abstraction)
Testing Framework	`pytest` with `pytest-asyncio` for async tests	Same, plus `hypothesis` for property-based testing
Type Hints	Extensive use of `typing` module (`Dict` , `List` , `Optional` , etc.)	`pydantic` for data validation or `mypy` strict checking

B. Recommended File/Module Structure:

```
graphql-engine/
├── README.md
├── pyproject.toml
├── src/
│   └── graphql_engine/
│       ├── __init__.py
│       ├── exceptions.py          # Custom exception types
│       │
│       ├── parsing/               # Milestone 1
│       │   ├── __init__.py
│       │   ├── lexer.py           # Tokenizer
│       │   ├── parser.py          # Recursive descent parser
│       │   ├── ast.py             # AST node classes
│       │   └── errors.py          # Parse error with location
│       │
│       ├── type_system/           # Milestone 2
│       │   ├── __init__.py
│       │   ├── types.py           # Base GraphQLType, ScalarType, ObjectType, etc.
│       │   ├── schema.py          # Schema class and validation
│       │   ├── introspection.py   # __schema, __type resolvers
│       │   └── values.py          # Value coercion utilities
│       │
│       ├── execution/             # Milestone 3
│       │   ├── __init__.py
│       │   ├── executor.py        # Main execute_query function
│       │   ├── resolver.py        # Resolver registry and calling
│       │   ├── errors.py          # GraphQLError collection
│       │   ├── dataloader.py      # Simple DataLoader implementation
│       │   └── context.py         # ExecutionContext class
│       │
│       ├── reflection/            # Milestone 4
│       │   ├── __init__.py
│       │   ├── adapter.py         # DatabaseAdapter base class
│       │   ├── postgres_adapter.py # PostgreSQL implementation
│       │   ├── mapper.py          # Table→Type, Column→Field mapping
│       │   ├── relationship.py    # Foreign key detection
│       │   └── naming.py          # Name conversion utilities
│       │
│       ├── compilation/           # Milestone 5
│       │   ├── __init__.py
│       │   ├── compiler.py        # AST-to-SQL compiler
│       │   ├── sql_builder.py     # SQL string builder with parameterization
│       │   ├── joins.py           # JOIN generation logic
│       │   ├── filters.py         # WHERE clause generation
│       │   └── pagination.py      # LIMIT/OFFSET handling
│       │
│       └── utils/
│           ├── __init__.py
│           ├── visitation.py      # AST visitor pattern
│           └── introspection.py   # Helper for schema inspection
│
└── tests/                         # Comprehensive test suite
    ├── __init__.py
    ├── test_parsing.py
    ├── test_type_system.py
    ├── test_execution.py
    ├── test_reflection.py
    ├── test_compilation.py
    └── integration/               # End-to-end tests
```

C. Infrastructure Starter Code:

Since the educational focus is on the GraphQL engine itself, here's complete starter code for a simple HTTP server and database adapter that learners can use immediately:

`src/graphql_engine/server/http_server.py`:

```python
"""
Simple HTTP server for testing GraphQL engine.
Educational purpose only - not production ready.
"""
import json
from http.server import HTTPServer, BaseHTTPRequestHandler
from typing import Dict, Any, Optional
from graphql_engine.parsing import parse_query
from graphql_engine.execution import execute_query
from graphql_engine.type_system import Schema


class GraphQLRequestHandler(BaseHTTPRequestHandler):
    """Handles POST requests with GraphQL queries in JSON body."""

    def __init__(self, schema: Schema, *args, **kwargs):
        self.schema = schema
        super().__init__(*args, **kwargs)

    def do_POST(self):
        """Handle GraphQL query POST request."""
        content_length = int(self.headers.get('Content-Length', 0))
        body = self.rfile.read(content_length).decode('utf-8')

        try:
            request_data = json.loads(body)
            query = request_data.get('query', '')
            variables = request_data.get('variables', {})
            operation_name = request_data.get('operationName')

            # Parse and execute
            document_ast = parse_query(query)
            result = execute_query(
                schema=self.schema,
                document_ast=document_ast,
                variable_values=variables,
                operation_name=operation_name
```

```
            )

            response_data = json.dumps(result, indent=2).encode('utf-8')

            self.send_response(200)
            self.send_header('Content-Type', 'application/json')
            self.send_header('Content-Length', str(len(response_data)))
            self.end_headers()
            self.wfile.write(response_data)

        except Exception as e:
            error_response = {
                'errors': [{'message': str(e)}]
            }
            response_data = json.dumps(error_response).encode('utf-8')

            self.send_response(400)
            self.send_header('Content-Type', 'application/json')
            self.send_header('Content-Length', str(len(response_data)))
            self.end_headers()
            self.wfile.write(response_data)

def start_test_server(schema: Schema, host: str = 'localhost', port: int = 8000):
    """Start a simple HTTP server for testing GraphQL queries."""
    def handler(*args, **kwargs):
        return GraphQLRequestHandler(schema, *args, **kwargs)

    server = HTTPServer((host, port), handler)
    print(f"GraphQL test server running at http://{host}:{port}")
    print("Send POST requests with JSON body: {{'query': '{{ ... }}', 'variables': {{...}}}}")
    server.serve_forever()
```

D. Core Logic Skeleton Code:

For the main execution function that learners will implement in Milestone 3:

`src/graphql_engine/execution/executor.py`:

PYTHON

```python
"""
Core query execution engine.
Implements GraphQL execution algorithm per specification.
"""
from typing import Dict, Any, Optional, List
from graphql_engine.type_system import Schema, ObjectType
from graphql_engine.parsing import Document, OperationDefinition
from graphql_engine.execution.context import ExecutionContext
from graphql_engine.execution.errors import GraphQLError


def execute_query(
    schema: Schema,
    document_ast: Document,
    variable_values: Optional[Dict[str, Any]] = None,
    operation_name: Optional[str] = None,
    context_value: Any = None
) -> Dict[str, Any]:
    """
    Execute a GraphQL query against a schema.

    Args:
        schema: The GraphQL schema to execute against
        document_ast: Parsed AST of the GraphQL query document
        variable_values: Dictionary of variable values provided with the request
        operation_name: Name of operation to execute if document contains multiple
        context_value: Shared context passed to all resolvers

    Returns:
        Dictionary containing 'data' and/or 'errors' keys per GraphQL spec

    TODO Implementation Steps (Milestone 3):
    1. Validate that document_ast contains at least one operation definition
    2. If operation_name is provided, find the named operation; otherwise use the only operation
    3. Coerce variable values to types defined in operation's variable definitions
    4. Create ExecutionContext with schema, variable values, and context_value
    5. Get root type (Query, Mutation, or Subscription) from schema based on operation type
```

```
    6. Execute selection set starting with root type and empty parent value
    7. Collect any errors that occurred during execution
    8. Return {'data': result} if no errors, or {'data': result, 'errors': [...]} if errors
    9. Ensure null propagation: if a non-null field resolves to null, bubble up appropriately
    10. Handle lists: if field type is List, execute for each item in parent value
    """
    # TODO 1: Validate document_ast has definitions
    # TODO 2: Select operation based on operation_name
    # TODO 3: Coerce variable values
    # TODO 4: Create ExecutionContext
    # TODO 5: Get root type from schema
    # TODO 6: Execute selection set (call execute_selection_set)
    # TODO 7: Collect errors from context
    # TODO 8: Format response according to GraphQL spec
    pass


def execute_selection_set(
    selection_set,
    parent_type: ObjectType,
    parent_value: Any,
    context: ExecutionContext
) -> Dict[str, Any]:
    """
    Execute a selection set for a given parent type and value.

    TODO Implementation Steps:
    1. Initialize result dict to hold field results
    2. For each selection in selection_set:
        a. If it's a Field: execute_field and add to result dict
        b. If it's a FragmentSpread: resolve fragment and execute its selection set
        c. If it's an InlineFragment: check type condition and execute if matches
    3. Return result dict
    4. Handle parallelism: sibling fields without dependencies can execute concurrently
    """
    pass
```

```
def execute_field(
    field,
    parent_type: ObjectType,
    parent_value: Any,
    context: ExecutionContext
) -> Any:
    """
    Execute a single field resolution.

    TODO Implementation Steps:
    1. Get field definition from parent_type
    2. Coerce field arguments using variable values from context
    3. Get resolver function for field (default: parent_value[field_name])
    4. Call resolver with (parent_value, args, context, info)
    5. If resolver returns None and field type is Non-Null: throw error
    6. If resolver returns a value and field has a selection set: recursively execute
    7. Return coerced value (apply List/NonNull wrappers as needed)
    """
    pass
```

E. Language-Specific Hints:

- **Python Type Hints:** Use `from typing import List, Dict, Optional, Any, Union` extensively. Define `TypedDict` for complex return structures.
- **Async/Await:** Consider using `asyncio` for concurrent field execution. Use `asyncio.gather()` for parallel sibling field resolution.
- **Error Handling:** Raise custom exceptions like `GraphQLError` with message and optional locations. Catch exceptions at field boundaries to allow partial results.
- **DataLoader Pattern:** Implement a simple `DataLoader` class that batches `load()` calls and caches results per request.

F. Milestone Checkpoint: After implementing the core execution engine (Milestone 3), run:

```bash
python -m pytest tests/test_execution.py -xvs
```

Expected output: All tests pass, demonstrating:

1. Simple field resolution returns correct data
2. Nested fields resolve recursively
3. Null propagation works correctly (non-null field resolving to null bubbles up)
4. List fields work with lists of objects
5. Field errors are collected without crashing execution

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
"Cannot read property 'x' of undefined" in resolver	Parent value not passed correctly to nested resolver	Print parent_value in execute_field	Ensure each field resolver receives the correct parent (object from previous level)
All fields return null	Resolver functions not registered or default resolver fails	Check field definition has resolver; add print to default resolver	Implement proper resolver lookup or ensure parent objects have matching attribute names
Non-null error bubbles to root incorrectly	Null propagation logic missing type checking	Add debug prints to see when null is encountered and what field type is	Implement spec rule: if field is NonNull and returns null, bubble null to parent; if parent is nullable, set to null
Sibling fields execute sequentially	Missing parallel execution	Add timing logs to see fields execute one after another	Use `asyncio.gather()` or `concurrent.futures` for independent fields

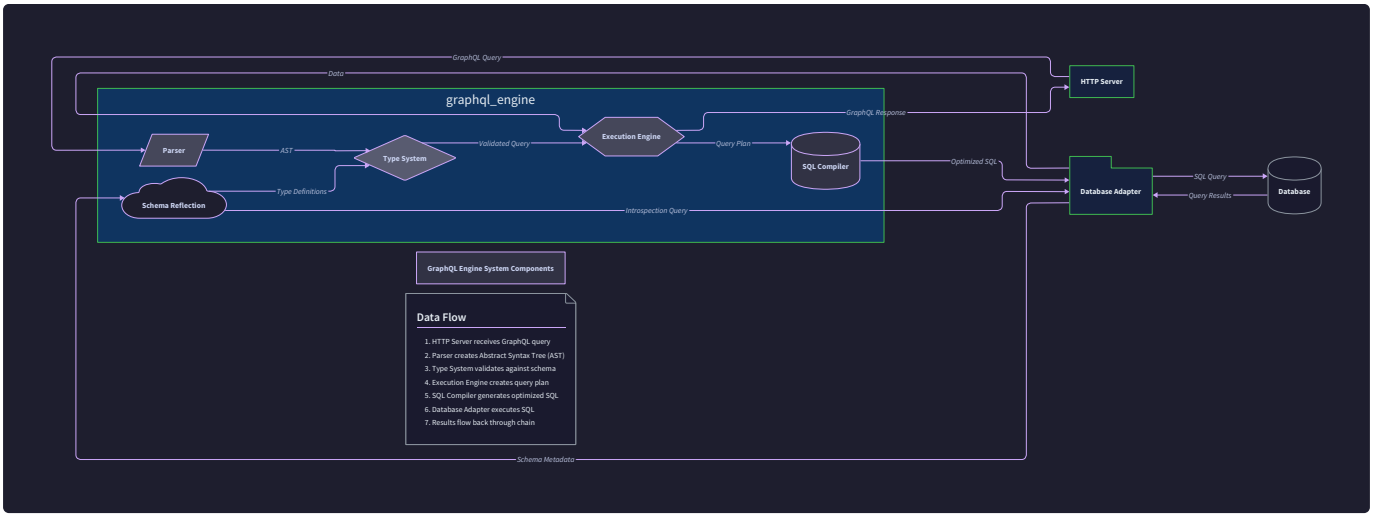
High-Level Architecture

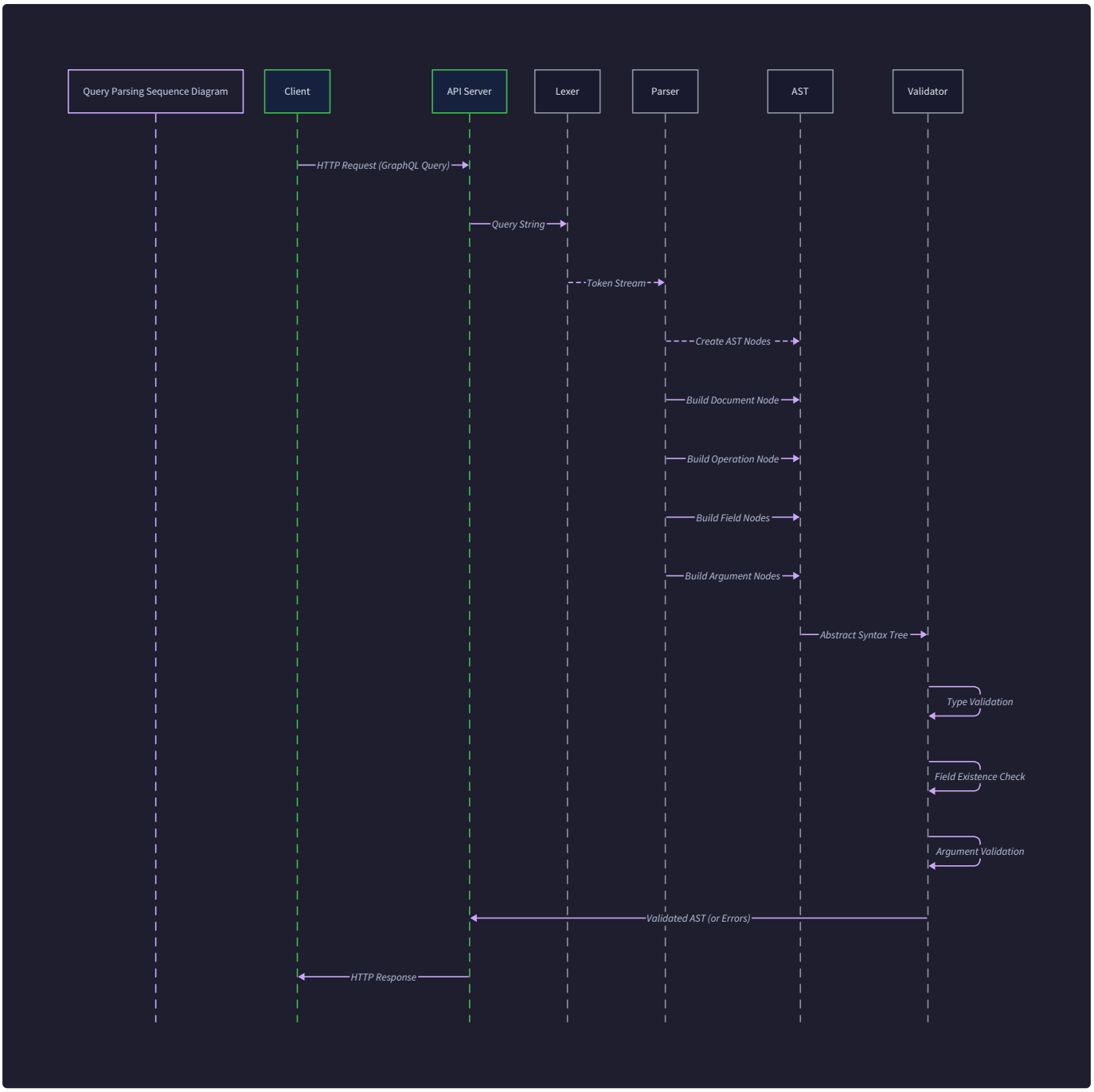
> **Milestone(s):** This section provides the architectural blueprint for all five milestones, showing how components fit together and how data flows through the system. It establishes the overall structure before diving into individual component details.

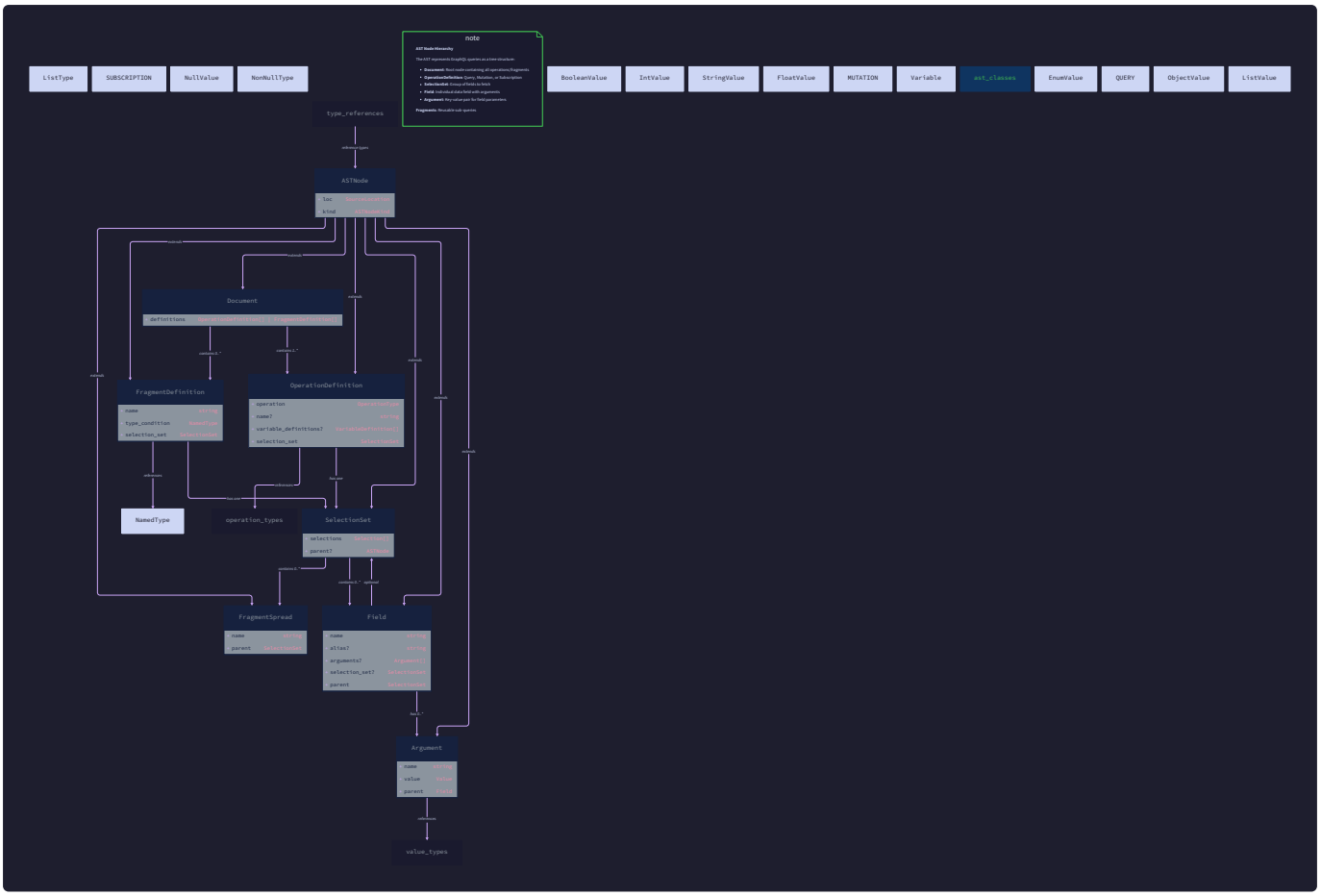
The GraphQL engine architecture bridges two fundamentally different worlds: GraphQL's hierarchical, client-centric query model and relational databases' tabular, set-oriented structure. This **impedance mismatch** is the central architectural challenge we must solve. Rather than a monolithic blob, we decompose the system into five focused components, each with a clear responsibility and interface, connected by well-defined data transformations.

System Component Diagram

> **Mental Model: The Restaurant Kitchen Analogy**
>
> Think of the GraphQL engine as a high-end restaurant kitchen. The **Parser** is the order-taker who listens to the customer's complex request and writes it down in the kitchen's internal shorthand (the AST). The **Type System** is the recipe book that defines what ingredients are available and how they can be combined. The **Execution Engine** is the head chef who coordinates the preparation, calling on various station chefs (resolvers) to prepare each component of the meal. The **Schema Reflection** is the inventory system that automatically catalogs what's in the pantry (database) and creates recipes for it. Finally, the **SQL Compiler** is the sous-chef who optimizes the cooking process—instead of making each component separately (N+1 problem), they plan an efficient sequence that uses shared ingredients and parallel cooking stations to prepare everything in minimal time.




Query Parsing Sequence Diagram
Client
API Server
Lexer
Parser
AST
Validator
HTTP Request (GraphQL Query)
Query String
Token Stream
Create AST Nodes
Build Document Node
Build Operation Node
Build Field Nodes
Build Argument Nodes
Abstract Syntax Tree
Type Validation
Field Existence Check
Argument Validation
Validated AST (or Errors)
HTTP Response



The system comprises five core components that transform a GraphQL query string into database results:

Component	Responsibility	Input	Output	Analogy
GraphQL Parser	Converts GraphQL query strings into Abstract Syntax Trees (ASTs)	GraphQL query string (`String`)	Validated AST (`Document`)	Order-taker translating customer request to kitchen ticket
Schema & Type System	Defines and validates GraphQL types, provides introspection	Type definitions, database metadata	Complete `GraphQLSchema` with all types	Recipe book defining available dishes and ingredients
Query Execution Engine	Executes queries by calling resolver functions	`GraphQLSchema` , query AST, variables, context	Structured JSON response	Head chef coordinating station chefs to prepare meal
Database Schema Reflection	Auto-generates GraphQL types from database schema	Database connection, configuration options	Generated GraphQL types and resolvers	Inventory system cataloging pantry contents
Query to SQL Compiler	Compiles GraphQL queries to optimized SQL	Query AST, schema metadata, variable values	Parameterized SQL query + execution plan	Sous-chef optimizing cooking process to minimize steps

Component Interactions and Data Flow

The components interact through well-defined interfaces, creating a pipeline that transforms data at each stage:

1. **Schema Building Path**: The Database Schema Reflection component reads database metadata and generates GraphQL types, which feed into the Schema & Type System component to build a complete schema.
2. **Query Execution Path**: A GraphQL query string flows through the Parser → Type System (for validation) → Execution Engine → SQL Compiler (for database queries) → Database → Back through the chain as results.

3. **Introspection Path**: Special GraphQL introspection queries follow the same execution path but are handled by built-in resolvers in the Type System component.

> **Architecture Decision: Separation of Concerns via Pipeline**
>
> **Context**: We need to transform GraphQL queries into database results while maintaining clarity, testability, and the ability to evolve components independently. The system must handle both hand-written schemas and auto-generated ones from databases.
>
> **Options Considered**:
>
> 1. **Monolithic Engine**: Single component that handles parsing, validation, execution, and SQL generation in one intertwined codebase.
> 2. **Pipeline Architecture**: Clear separation with each component focusing on one transformation, passing data through well-defined interfaces.
> 3. **Plugin Architecture**: Core engine with extensible plugins for parsing, execution, SQL generation, etc.
>
> **Decision**: Pipeline Architecture with clear interfaces between components.
>
> **Rationale**:
>
> - **Testability**: Each component can be tested in isolation with mock inputs/outputs.
> - **Educational Value**: Clear boundaries help learners understand each transformation step.
> - **Evolution**: Components can be swapped or upgraded independently (e.g., different SQL dialects).
> - **Debugging**: Failures can be pinpointed to specific transformation stages.
>
> **Consequences**:
>
> - **Positive**: Clear mental model, easier to understand and debug, enables incremental development.
> - **Negative**: Some overhead in data marshaling between components, requires careful interface design.
> - **Mitigation**: Use immutable data structures that can be passed efficiently by reference.

Option	Pros	Cons	Chosen?
Monolithic Engine	Fewer interface boundaries, potentially faster	Hard to test, difficult to understand, tightly coupled	❌
Pipeline Architecture	Clear separation, testable, educational, modular	Data marshaling overhead, interface design complexity	✅
Plugin Architecture	Extremely flexible, ecosystem potential	Complex plugin API, harder for learners, over-engineering	❌

Recommended File/Module Structure

> **Mental Model: The Workshop Organization**
>
> Imagine each component as a specialized workshop station. The **Parser Workshop** has lexing tools and grammar charts. The **Type System Workshop** has blueprint diagrams and validation checklists. The **Execution Workshop** has assembly lines and quality control stations. The **Reflection Workshop** has database connection tools and mapping tables. The **SQL Compiler Workshop** has query planning boards and optimization tools. Each workshop has its own tools, organized in labeled drawers (modules), with clear paths between workshops for moving work-in-progress.

A clean directory structure is critical for managing complexity and maintaining separation of concerns. Below is the recommended organization for a Python implementation:

```
graphql-engine/
├── pyproject.toml             # Project configuration and dependencies
├── README.md
├── examples/                  # Example schemas and queries
│   ├── blog/
│   └── ecommerce/
├── src/
│   └── graphql_engine/        # Main package
│       ├── __init__.py
│       ├── exceptions.py      # Custom exception types
│       ├── constants.py       # Constants (type names, directives, etc.)
│       ├── parser/            # Component 1: GraphQL Parser
│       │   ├── __init__.py
│       │   ├── lexer.py       # Tokenizer/Lexer
│       │   ├── parser.py      # Recursive descent parser
│       │   ├── ast.py         # AST node class definitions
│       │   └── errors.py      # Parse error types
│       ├── type_system/       # Component 2: Schema & Type System
│       │   ├── __init__.py
│       │   ├── types.py       # Base GraphQLType and all type classes
│       │   ├── schema.py      # GraphQLSchema class and validation
│       │   ├── introspection.py # Introspection resolver implementations
│       │   └── validation.py  # Schema validation rules
│       ├── execution/         # Component 3: Query Execution Engine
│       │   ├── __init__.py
│       │   ├── executor.py    # Main Executor class
│       │   ├── resolvers.py   # Default resolver implementations
│       │   ├── dataloader.py  # DataLoader pattern implementation
│       │   ├── errors.py      # Execution error handling
│       │   └── context.py     # ExecutionContext class
│       ├── reflection/        # Component 4: Database Schema Reflection
│       │   ├── __init__.py
│       │   ├── reflector.py   # Main SchemaReflector class
│       │   ├── adapters/      # Database-specific adapters
│       │   │   ├── __init__.py
│       │   │   ├── postgres.py
│       │   │   ├── mysql.py
│       │   │   └── sqlite.py
│       │   ├── type_mapping.py # SQL-to-GraphQL type mapping
│       │   └── naming.py      # Naming convention utilities
│       ├── sql_compiler/      # Component 5: Query to SQL Compilation
│       │   ├── __init__.py
│       │   ├── compiler.py    # Main SQLCompiler class
│       │   ├── ir.py          # Intermediate Representation structures
│       │   ├── builders.py    # SQL query builder utilities
│       │   ├── joins.py       # JOIN generation logic
│       │   └── parameters.py  # Query parameterization
│       └── server/            # Optional HTTP server wrapper
│           ├── __init__.py
│           ├── http_handler.py # HTTP request/response handling
│           └── graphiql.py    # GraphiQL IDE endpoint
└── tests/                     # Comprehensive test suite
    ├── __init__.py
    ├── test_parser/
    ├── test_type_system/
    ├── test_execution/
    ├── test_reflection/
    └── test_sql_compiler/
```

Module Responsibilities and Public Interfaces

Each module exposes a clean public API while hiding implementation details:

Module	Public Classes/Functions	Purpose	Dependencies
`parser`	`parse_query(query_str)`, `parse_schema(sdl_str)`, `Document`, `Location`	Convert strings to ASTs	None (pure text processing)
`type_system`	`GraphQLSchema`, `GraphQLObjectType`, `GraphQLScalarType`, `build_schema()`	Define and validate types	`parser` for SDL parsing
`execution`	`execute_query(schema, document, ...)`, `ExecutionContext`, `DataLoader`	Execute queries against schema	`parser`, `type_system`
`reflection`	`SchemaReflector`, `reflect_schema(connection)`, database adapters	Generate schema from database	`type_system` for type building
`sql_compiler`	`SQLCompiler`, `compile_to_sql(query_ast, schema)`, `CompiledQuery`	Convert GraphQL to SQL	`parser`, `type_system`, `reflection`

Architecture Decision: Layered Dependencies

Context: Components have natural dependencies (e.g., Execution needs Parser and Type System, SQL Compiler needs all previous components). We must manage these dependencies to avoid circular references and ensure clean separation.

Options Considered:

1. **Flat Structure**: All components in same module level with bidirectional imports.
2. **Layered Architecture**: Lower layers don't know about higher layers (Parser → Type System → Execution → Reflection/SQL Compiler).
3. **Dependency Injection**: All components receive dependencies via constructor injection.

Decision: Layered Architecture with unidirectional dependencies.

Rationale:

- **Clear Dependency Flow**: Natural progression from parsing → typing → execution → compilation.
- **No Circular Dependencies**: Prevents import loops and spaghetti code.
- **Progressive Learning**: Learners can build and test components in dependency order.
- **Simplified Testing**: Lower layers can be tested without higher layers.

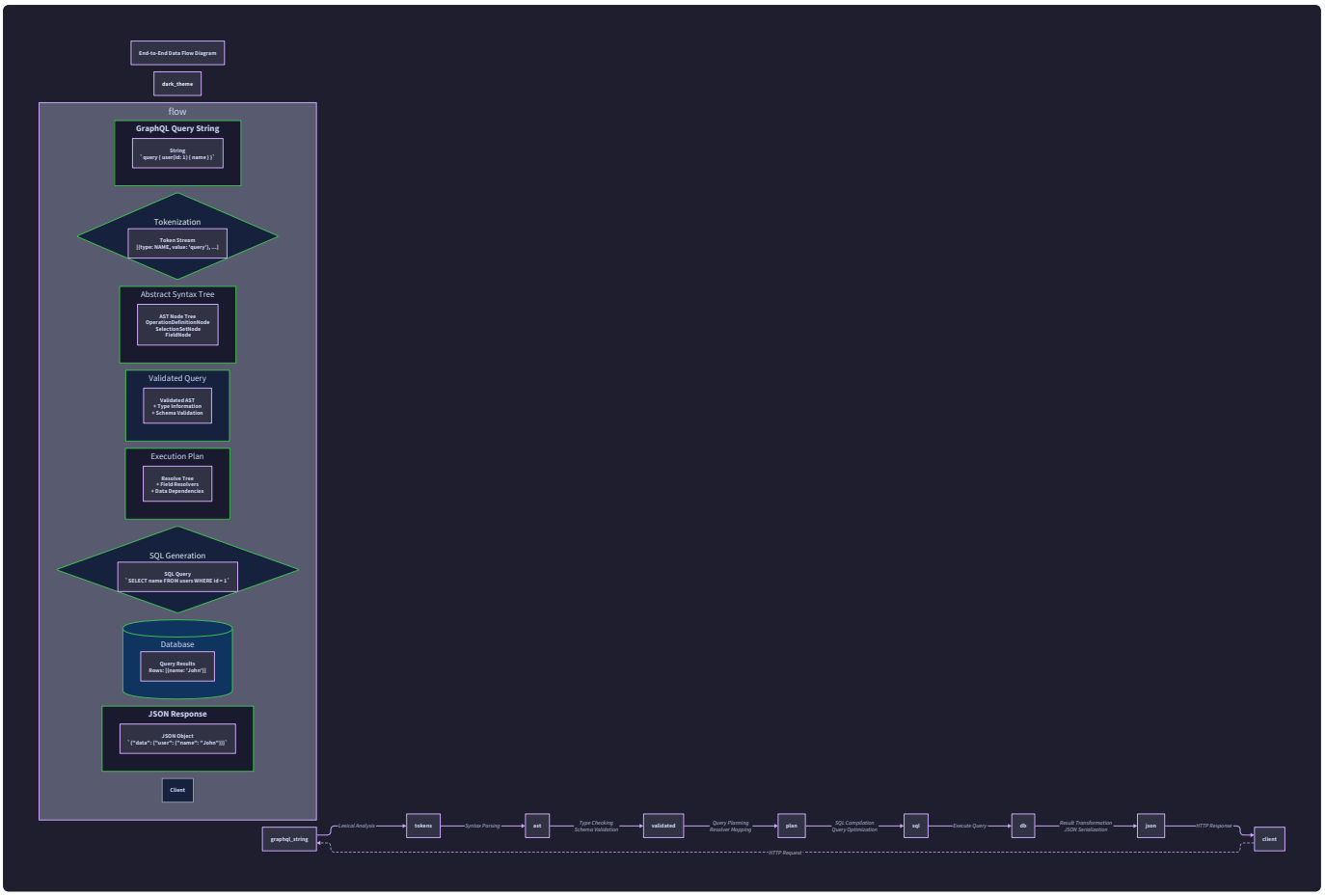
Consequences:

- **Positive**: Clean mental model, predictable dependencies, easier to reason about.
- **Negative**: Higher layers can't easily call back to lower layers (though rarely needed).
- **Mitigation**: Use callback interfaces when upward communication is genuinely needed.

End-to-End Data Flow

Mental Model: The Manufacturing Assembly Line

Picture a car manufacturing plant where raw materials (GraphQL query strings) enter at one end and finished cars (JSON responses) exit at the other. At Station 1 (Parser), the query sheet metal is cut and shaped into standardized parts (AST nodes). At Station 2 (Type System), each part is inspected against blueprints to ensure it fits specifications. At Station 3 (Execution), the assembly line coordinates workers (resolvers) to assemble components. For database-backed fields, instead of hand-crafting each part, Station 4 (SQL Compiler) designs efficient machine tools (SQL queries) that mass-produce precisely the needed components. Station 5 (Database) is the raw materials warehouse that feeds these machines. Finally, quality control assembles everything into the final product.



The complete transformation pipeline for a typical query involves eight distinct data format transformations:

Step-by-Step Data Transformation Pipeline

1. **Input Reception**:
   - **Format**: HTTP POST request with JSON body
   - **Content**: `{"query": "query { user(id: 1) { name email } }", "variables": {...}, "operationName": "..."}`
   - **Source**: Client application (e.g., React frontend, mobile app)
2. **GraphQL Query String Extraction**:
   - **Action**: Extract the `query` field from the JSON payload
   - **Format**: Raw GraphQL query string
   - **Example**: `"query GetUser($id: ID!) { user(id: $id) { name email posts { title } } }"`
   - **Note**: Variables are kept separate for later substitution
3. **Lexical Analysis (Tokenizer)**:
   - **Action**: Break string into tokens with type and position
   - **Input**: `"query { user(id: 1) { name } }"`
   - **Output**: List of `Token` objects:

Token Type	Value	Location (line:column)
NAME	"query"	1:1
BRACE_L	"{"	1:7
NAME	"user"	1:9
PAREN_L	"("	1:13
NAME	"id"	1:14
COLON	":"	1:16
INT	"1"	1:18
PAREN_R	")"	1:19
BRACE_L	"{"	1:21
NAME	"name"	1:23
BRACE_R	"}"	1:27
BRACE_R	"}"	1:29

4. **Syntactic Analysis (Parser)**:
   - **Action**: Build AST from token stream using recursive descent
   - **Output**: `Document` AST with `OperationDefinition`, `SelectionSet`, `Field`, `Argument` nodes
   - **Structure**:

```
Document
└── OperationDefinition (type: "query", name: None)
    └── SelectionSet
        └── Field (name: "user")
            ├── Argument (name: "id", value: IntValue(1))
            └── SelectionSet
                └── Field (name: "name")
```

5. **Schema Validation**:
   - **Action**: Check AST against schema type definitions
   - **Checks**: Field existence, argument type compatibility, required arguments provided
   - **Output**: Validated query AST or detailed validation errors
   - **Critical Validation Rules**:
     1. All selected fields must exist on the parent type
     2. All required arguments must be provided
     3. Variable types must match parameter types
     4. Fragments must be used on compatible types

6. **Execution Planning**:
   - **Action**: Analyze query to determine resolution strategy
   - **For Database Fields**: Pass to SQL Compiler
   - **For Custom Resolvers**: Schedule resolver execution
   - **Output**: Execution plan with parallelizable units

7. **SQL Compilation** (for database-backed fields):
   - **Action**: Convert GraphQL field selections to SQL query
   - **Input**: `Field` AST nodes, schema type information, variable values
   - **Process**:
     1. Analyze selection tree to determine required tables and columns
     2. Generate JOINs based on foreign key relationships

3. Apply WHERE clauses from arguments
4. Add ORDER BY/LIMIT for pagination

- **Output**: Parameterized SQL query + bind parameters

```sql
SELECT "user"."id", "user"."name", "user"."email"
FROM "users" AS "user"
WHERE "user"."id" = $1  -- $1 = 1
```

8. **Database Query Execution**:
   - **Action**: Execute compiled SQL with parameters
   - **Format**: Database-specific query + parameter binding
   - **Output**: Raw database rows (list of dictionaries/records)
9. **Result Shaping**:
   - **Action**: Transform database rows to GraphQL response shape
   - **Process**: Map column values to field names, nest related objects
   - **Output**: Nested dictionaries matching query structure

```json
{
  "user": {
    "name": "Alice",
    "email": "alice@example.com"
  }
}
```

10. **Error Collection & Final Response Assembly**:
    - **Action**: Gather any field errors, apply error masking if needed
    - **Output**: Final GraphQL response JSON:

```json
{
  "data": { ... },
  "errors": [ ... ]  // Only if errors occurred
}
```

Concrete Walk-Through Example

Let's trace a complete example query through the pipeline:

Initial HTTP Request:

```http
POST /graphql HTTP/1.1
Content-Type: application/json

{
  "query": "query GetUserWithPosts($userId: ID!) { user(id: $userId) { name email posts(limit: 5) { title publishedAt } }
}",
  "variables": { "userId": "42" },
  "operationName": "GetUserWithPosts"
}
```

Step 1-2: Extract query string and variables:

- Query: `"query GetUserWithPosts($userId: ID!) { user(id: $userId) { name email posts(limit: 5) { title publishedAt } } }"`
- Variables: `{"userId": "42"}`

Step 3-4: Parser produces AST:

```
Document
├── OperationDefinition
│   ├── operation_type: "query"
│   ├── name: "GetUserWithPosts"
│   ├── variable_definitions:
│   │   └── VariableDefinition
│   │       ├── name: "userId"
│   │       └── type: NonNullType(of: NamedType(name: "ID"))
│   └── selection_set: SelectionSet
│       └── Field
│           ├── name: "user"
│           ├── arguments:
│           │   └── Argument
│           │       ├── name: "id"
│           │       └── value: Variable(name: "userId")
│           └── selection_set: SelectionSet
│               ├── Field(name: "name")
│               ├── Field(name: "email")
│               └── Field(name: "posts")
│                   ├── arguments:
│                   │   └── Argument
│                   │       ├── name: "limit"
│                   │       └── value: IntValue(value: 5)
│                   └── selection_set: SelectionSet
│                       ├── Field(name: "title")
│                       └── Field(name: "publishedAt")
```

Step 5: Type System validates:

- `User` type has fields `name`, `email`, `posts`
- `posts` field returns `[Post]` type
- `Post` type has fields `title`, `publishedAt`
- `user` argument `id` expects `ID!` → variable `userId` is `ID!` ✓
- `posts` argument `limit` expects `Int` → value `5` is `Int` ✓

Step 6-7: SQL Compiler analyzes and generates:

- Need `users` table columns: `id`, `name`, `email`
- Need `posts` table columns: `id`, `title`, `published_at`, `author_id`
- Relationship: `posts.author_id` → `users.id`
- Generate SQL with JOIN:

```sql
SELECT
  "user"."id" AS "user__id",
  "user"."name" AS "user__name",
  "user"."email" AS "user__email",
  "posts"."id" AS "posts__id",
  "posts"."title" AS "posts__title",
  "posts"."published_at" AS "posts__published_at"
FROM "users" AS "user"
LEFT JOIN LATERAL (
  SELECT "posts".*
  FROM "posts"
  WHERE "posts"."author_id" = "user"."id"
  ORDER BY "posts"."published_at" DESC
  LIMIT 5
) AS "posts" ON TRUE
WHERE "user"."id" = $1  -- $1 = 42
```

Step 8: Database returns rows:

```
[
  {
    "user__id": 42,
    "user__name": "Alice",
    "user__email": "alice@example.com",
    "posts__id": 101,
    "posts__title": "GraphQL Tutorial",
    "posts__published_at": "2023-10-01 14:30:00"
  },
  {
    "user__id": 42,
    "user__name": "Alice",
    "user__email": "alice@example.com",
    "posts__id": 102,
    "posts__title": "Database Design",
    "posts__published_at": "2023-10-05 09:15:00"
  }
]
```

Step 9: Result shaping transforms to nested structure:

```json
{
  "user": {
    "name": "Alice",
    "email": "alice@example.com",
    "posts": [
      {
        "title": "GraphQL Tutorial",
        "publishedAt": "2023-10-01T14:30:00Z"
      },
      {
        "title": "Database Design",
        "publishedAt": "2023-10-05T09:15:00Z"
      }
    ]
  }
}
```

Step 10: Final response assembly (no errors):

```json
{
  "data": {
    "user": { ... }  // as above
  }
}
```

Data Formats at Each Stage

Stage	Data Format	Key Properties	Example
1. Client Request	HTTP + JSON	Contains query string, variables, operation name	`{"query": "{ user { name } }", "variables": {}}`
2. Extracted Query	GraphQL String	Raw query text with possible variables	`"query { user(id: 1) { name } }"`
3. Token Stream	List of `Token`	Linear sequence with type, value, location	`[Token(NAME, "query", 1:1), Token(BRACE_L, "{", 1:7), ...]`
4. AST	`Document` object	Tree structure with nodes preserving source locations	`Document(definitions=[OperationDefinition(...)])`
5. Validated AST	`Document` + type info	AST annotated with resolved types	Field node knows it resolves to `User` type
6. Execution Plan	`ExecutionPlan`	Graph of resolvers with dependencies	`[SQLResolver(table="users"), CustomResolver(field="bio")]`
7. Compiled SQL	`CompiledQuery`	SQL string + parameter bindings	`("SELECT ... WHERE id = $1", [42])`
8. Database Results	List of rows	Tabular data, possibly with aliased columns	`[{"user__name": "Alice", "posts__title": "..."}]`
9. Shaped Results	Nested dict	Hierarchical structure matching query	`{"user": {"name": "Alice", "posts": [...]}}`
10. Final Response	GraphQL Response JSON	Standard GraphQL response format	`{"data": ..., "errors": [...]}`

> **Key Insight**: Each transformation makes the data more specialized for the next stage's needs. The AST is optimized for tree traversal, the SQL is optimized for database execution, and the final JSON is optimized for client consumption. This separation allows each component to use the most appropriate data structure for its task.

Common Pitfalls in Data Flow Design

⚠️ Pitfall: Premature SQL Generation

- **Description**: Generating SQL immediately after parsing, before understanding the full query structure.
- **Why Wrong**: Cannot optimize across nested relationships, leads to N+1 queries.
- **Fix**: First build complete execution plan analyzing all nested selections, then generate optimized SQL with JOINs.

⚠️ Pitfall: Losing Location Information

- **Description**: Not preserving source locations through the pipeline.
- **Why Wrong**: Error messages cannot point to the exact location in the original query.
- **Fix**: Every AST node should store its `Location` (line, column) and propagate it through transformations.

⚠️ Pitfall: Mixing Validation and Execution

- **Description**: Performing validation during execution (e.g., checking field existence when resolving).
- **Why Wrong**: Errors surface late, performance overhead on every execution.
- **Fix**: Complete all validation before execution begins in a dedicated validation phase.

⚠️ Pitfall: Inefficient Result Shaping

- **Description**: Transforming database rows to JSON by iterating over the AST for each row.
- **Why Wrong**: O(n × m) complexity for n rows and m fields.
- **Fix**: Pre-compute transformation mapping from column aliases to response paths, apply once per row.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Learning Focus)	Advanced Option (Production Ready)
HTTP Server	Python `http.server` module	FastAPI or Starlette with async support
Parser Implementation	Hand-written recursive descent parser	Generated parser using ANTLR or Lark
Type Representation	Python classes with dict-based fields	Immutable data classes with slots
Database Access	SQLite with `sqlite3` module	PostgreSQL with `asyncpg` or SQLAlchemy
SQL Building	String concatenation with f-strings	SQLAlchemy Core or custom query builder
Async Execution	Synchronous execution (simpler)	Asyncio with concurrent resolvers
Testing	`unittest` with mock databases	`pytest` with test containers

B. Recommended Starter Project Structure

For learners starting the implementation, begin with this minimal structure:

```
my-graphql-engine/
├── src/
│   └── graphql_engine/
│       ├── __init__.py
│       ├── parser.py        # Start with parser (Milestone 1)
│       ├── types.py         # Add for Milestone 2
│       ├── execution.py     # Add for Milestone 3
│       ├── reflection.py    # Add for Milestone 4
│       └── sql_compiler.py  # Add for Milestone 5
└── tests/
    ├── test_parser.py
    ├── test_types.py
    └── ...
```

C. Infrastructure Starter Code

Here's complete starter code for the HTTP server layer (not the core learning goal but needed to run the engine):

PYTHON

```python
# src/graphql_engine/server/http_handler.py

"""
HTTP handler for GraphQL requests. This is infrastructure code that learners
can use as-is without implementing the HTTP layer themselves.
"""
import json
from http.server import BaseHTTPRequestHandler, HTTPServer
from typing import Dict, Any, Optional
import traceback


class GraphQLHTTPHandler(BaseHTTPRequestHandler):
    """Handles HTTP POST requests with GraphQL queries."""

    def __init__(self, graphql_executor, *args, **kwargs):
        """Initialize with a GraphQL executor function."""
        self.graphql_executor = graphql_executor
        super().__init__(*args, **kwargs)

    def do_POST(self):
        """Handle POST requests to /graphql endpoint."""
        if self.path != '/graphql':
            self.send_response(404)
            self.end_headers()
            return

        content_length = int(self.headers.get('Content-Length', 0))
        if content_length == 0:
            self.send_response(400)
            self.end_headers()
            self.wfile.write(b'{"error": "Empty request body"}')
            return

        try:
            # Read and parse JSON request
            body = self.rfile.read(content_length)
            request_data = json.loads(body)
```

```
        # Extract GraphQL query components
        query = request_data.get('query')
        variables = request_data.get('variables') or {}
        operation_name = request_data.get('operationName')

        if not query:
            raise ValueError("Missing 'query' field in request")

        # Execute GraphQL query
        result = self.graphql_executor(
            query=query,
            variables=variables,
            operation_name=operation_name
        )

        # Send successful response
        self.send_response(200)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(result).encode('utf-8'))

    except json.JSONDecodeError:
        self.send_response(400)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps({
            "error": "Invalid JSON in request body"
        }).encode('utf-8'))

    except Exception as e:
        # Internal server error
        self.send_response(500)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        error_response = {
```

```
            "error": str(e),
            "traceback": traceback.format_exc() if self.debug else None
        }
        self.wfile.write(json.dumps(error_response).encode('utf-8'))

def do_GET(self):
    """Serve GraphiQL interface at root path."""
    if self.path == '/' or self.path == '/graphiql':
        self.send_response(200)
        self.send_header('Content-Type', 'text/html')
        self.end_headers()
        self.wfile.write(self._graphiql_html().encode('utf-8'))
    else:
        self.send_response(404)
        self.end_headers()

def _graphiql_html(self) -> str:
    """Return HTML for GraphiQL interface."""
    return """
    <!DOCTYPE html>
    <html>
    <head>
        <title>GraphiQL</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/graphiql/0.17.5/graphiql.min.css">
    </head>
    <body>
        <div id="graphiql" style="height: 100vh;"></div>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/react/16.8.0/umd/react.production.min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/react-dom/16.8.0/umd/react-dom.production.min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/graphiql/0.17.5/graphiql.min.js"></script>
        <script>
            function graphQLFetcher(params) {
                return fetch('/graphql', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json',
```

```
                        },
                        body: JSON.stringify(params),
                    }).then(response => response.json());
                }
                ReactDOM.render(
                    React.createElement(GraphiQL, { fetcher: graphQLFetcher }),
                    document.getElementById('graphiql')
                );
            </script>
        </body>
        </html>
        """

    def log_message(self, format, *args):
        """Override to control logging output."""
        # Optional: Implement logging to file or stdout
        print(f"{self.address_string()} - {format % args}")

def start_server(executor, host='localhost', port=8000, debug=False):
    """
    Start HTTP server with GraphQL endpoint.

    Args:
        executor: Function that accepts (query, variables, operation_name)
                  and returns GraphQL response
        host: Server hostname
        port: Server port
        debug: Enable debug mode with tracebacks in errors
    """
    def handler(*args, **kwargs):
        return GraphQLHTTPHandler(executor, *args, **kwargs)

    handler.debug = debug

    server = HTTPServer((host, port), handler)
    print(f"GraphQL server running at http://{host}:{port}/")
```

```
    print(f"GraphiQL interface at http://{host}:{port}/graphiql")


    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nShutting down server...")
        server.server_close()
```

D. Core Pipeline Skeleton Code

Here's the main execution pipeline that learners will implement component by component:

PYTHON

```python
# src/graphql_engine/__init__.py

"""
Main GraphQL engine entry point. This orchestrates all components.
"""

from typing import Dict, Any, Optional
from .parser import parse_query
from .type_system import GraphQLSchema, validate_query
from .execution import execute_query as execute_query_internal
from .reflection import reflect_schema_from_db
from .sql_compiler import SQLCompiler


class GraphQLEngine:
    """Main GraphQL engine coordinating all components."""

    def __init__(self, schema: Optional[GraphQLSchema] = None):
        """
        Initialize engine with optional schema.

        Args:
            schema: Pre-built GraphQL schema. If None, can be built from DB.
        """
        self.schema = schema
        self.sql_compiler = SQLCompiler() if schema else None

    def execute(
        self,
        query: str,
        variables: Optional[Dict[str, Any]] = None,
        operation_name: Optional[str] = None,
        context: Optional[Dict[str, Any]] = None
    ) -> Dict[str, Any]:
        """
        Execute a GraphQL query through the complete pipeline.

        Args:
            query: GraphQL query string
```

```
        variables: Dictionary of variable values
        operation_name: Name of operation to execute (if multiple in query)
        context: Execution context (auth, dataloaders, etc.)

    Returns:
        GraphQL response dictionary with 'data' and/or 'errors' keys

    Raises:
        GraphQLError: For parsing, validation, or execution errors
    """
    # TODO 1: Parse query string into AST
    #   Call parse_query() to get Document AST
    #   Handle parse errors with proper error formatting

    # TODO 2: Validate query against schema
    #   Call validate_query() with schema and parsed AST
    #   Collect all validation errors (don't stop on first error)

    # TODO 3: If validation passes, execute query
    #   Call execute_query_internal() with schema, AST, variables, context
    #   This will internally use SQL compiler for database fields

    # TODO 4: Format response according to GraphQL spec
    #   Ensure response has 'data' key (even if null)
    #   Include 'errors' key only if errors occurred
    #   Preserve error paths and locations

    pass

def reflect_schema_from_database(
    self,
    connection_string: str,
    dialect: str = "postgresql"
) -> None:
    """
    Generate GraphQL schema from database metadata.
```

```
    Args:
        connection_string: Database connection string
        dialect: Database dialect ('postgresql', 'mysql', 'sqlite')
    """
    # TODO 1: Connect to database using appropriate adapter
    #   Use connection_string to establish database connection

    # TODO 2: Introspect database metadata
    #   Query information_schema for tables, columns, foreign keys
    #   Detect relationships and primary keys

    # TODO 3: Generate GraphQL types
    #   Create GraphQLObjectType for each table
    #   Map SQL types to GraphQL scalar types
    #   Create fields for relationships

    # TODO 4: Build complete schema
    #   Create Query type with root fields for each table
    #   Add mutations for create/update/delete operations
    #   Register custom resolvers for relationship fields

    # TODO 5: Initialize SQL compiler with schema
    #   Create SQLCompiler instance
    #   Pass table/relationship metadata for query planning

    pass
```

E. Language-Specific Hints for Python

- **Use Python's `enum.Enum`** for token types and AST node types for type safety
- **Use `@dataclass` decorator** for AST nodes to automatically get `__init__`, `__repr__`, and equality methods
- **Use `typing` module extensively** for type hints—helps catch errors early and serves as documentation
- **For recursive structures** (like ASTs), use forward references with string literals: `'Node'` instead of `Node`
- **Use Python's `contextvars`** for execution context that propagates through async calls
- **For SQL parameterization**, always use query parameters (`?` or `%s`) never string formatting to prevent injection
- **Use `asyncio.gather()`** for parallel resolver execution when implementing async version

F. Milestone Checkpoint for High-Level Architecture

After setting up the project structure (before implementing any components):

1. **Verify Structure**: Run `tree my-graphql-engine/` (or equivalent) to confirm directory structure matches recommendations.
2. **Test HTTP Starter**: Create a simple test file to verify the HTTP server works:

```python
# test_server.py

from src.graphql_engine.server.http_handler import start_server


def dummy_executor(query, variables, operation_name):

    return {"data": {"test": "Hello GraphQL"}}


if __name__ == "__main__":

    # Should start server on localhost:8000

    start_server(dummy_executor, debug=True)
```

PYTHON

- Expected: Server starts, visit `http://localhost:8000/graphiql` shows GraphiQL interface
- Send test query `{ test }` should return `{"data": {"test": "Hello GraphQL"}}`

3. **Import Check**: Create `src/graphql_engine/__init__.py` with the skeleton `GraphQLEngine` class above. Verify no syntax errors:

```bash
python -m py_compile src/graphql_engine/__init__.py
```

BASH

Signs of Problems:

- ❌ "ModuleNotFoundError: No module named 'graphql_engine'" → Check PYTHONPATH or install package in development mode (`pip install -e .`)
- ❌ HTTP server fails to start → Port 8000 might be in use, try `port=8001`
- ❌ GraphiQL interface doesn't load → Check internet connection (CDN resources) or use local GraphiQL build

G. Debugging Tips for Architecture Issues

Symptom	Likely Cause	How to Diagnose	Fix
Circular import error	Components importing each other	Check import statements in each module	Reorganize to follow layered dependencies (Parser → Type System → Execution)
Memory usage grows with queries	AST nodes not being garbage collected	Use `tracemalloc` to track allocations	Ensure AST nodes don't hold references to execution state
Response missing nested data	Result shaping not handling joins properly	Log raw database rows and compare to expected structure	Check SQL compiler column aliasing and result mapping logic
Queries slow with depth	Generating separate SQL per level instead of joins	Enable SQL logging and count queries	Implement join detection in SQL compiler for nested fields
Can't find database tables	Reflection querying wrong schema	Log the exact SQL used for introspection	Check database permissions and schema name in connection

Data Model

> **Milestone(s):** This section establishes the foundational data structures that underpin all five milestones. The AST node types (Milestone 1) capture the parsed query, the type system types (Milestone 2) define the schema, the execution context (Milestone 3) manages runtime state, and the SQL intermediate representation (Milestone 5) bridges the GraphQL and SQL worlds.

The data model is the backbone of our GraphQL engine—it defines how we represent GraphQL queries, schemas, execution state, and database operations in memory. Think of it as the **blueprint language** that all components use to communicate. If the components were departments in a company (parsing, execution, compilation), the data model would be the standard forms and documents they exchange.

This section details four core families of data structures:

1. **AST Node Types**: The parsed, in-memory tree representation of a GraphQL query string.
2. **Type System Types**: The definitions of allowed data shapes (scalars, objects, etc.) that constitute a GraphQL schema.
3. **Execution Context and State**: The runtime workspace that tracks progress, errors, and shared data during query resolution.
4. **SQL Intermediate Representation (IR)**: A bridge structure that captures the intent of a GraphQL query in SQL terms before generating the final SQL string.

AST Node Types

Mental Model: The Family Tree Think of a GraphQL query as a family tree. The `Document` is the entire family (the query document). `OperationDefinition` nodes are the main branches (query, mutation, subscription). Each branch has smaller twigs (`SelectionSet`), which hold leaves (`Field`). Some leaves have their own mini-branches (nested `SelectionSet`). Fragments are like sub-trees that can be grafted onto multiple branches. The AST is this entire family tree drawn on paper, with every member's name, location, and relationships explicitly recorded.

The Abstract Syntax Tree (AST) is the primary output of the parser (Milestone 1). It is a lossless, structured representation of the GraphQL query string that preserves every syntactic detail—field names, arguments, directives, fragment spreads, and their exact locations in the source text. This tree is what the execution engine and SQL compiler will later traverse.

All AST nodes inherit from a base `Node` type, which provides source location tracking. This is crucial for meaningful error messages (e.g., "Argument 'id' expected type 'ID!' at line 5, column 12").

Core AST Node Hierarchy and Data Structures

The following table details the complete hierarchy and fields of each AST node type. The inheritance relationship is: `Document` → `Definition` → `OperationDefinition` → `SelectionSet` → `Selection` → `Field` . Parallel branches exist for fragments and other definitions.

Type Name	Parent Type	Fields	Field Type	Description
`Location`	(none)	`line`	`int`	1-indexed line number in the source text.
		`column`	`int`	1-indexed column number in the source text.
`Node`	(none)	`loc`	`Optional[Location]`	Source location of this node. Present if the parser was configured to include it.
`Document`	`Node`	`definitions`	`List[Definition]`	List of operation and fragment definitions in this document.
`Definition`	`Node`	(none, base class)		Abstract base for definitions in a document.
`OperationDefinition`	`Definition`	`operation_type`	`str`	One of: "query", "mutation", "subscription".
		`name`	`Optional[str]`	Optional name of the operation (e.g., "GetUser").
		`variable_definitions`	`List[VariableDefinition]`	Variable declarations (`($id: ID!)`).
		`directives`	`List[Directive]`	Operation-level directives (e.g., `@deprecated`).
		`selection_set`	`SelectionSet`	The root selection set of fields to fetch.
`FragmentDefinition`	`Definition`	`name`	`str`	Name of the fragment (e.g., "UserFields").
		`type_condition`	`NamedType`	The type this fragment applies to (e.g., `User`).
		`directives`	`List[Directive]`	Fragment-level directives.
		`selection_set`	`SelectionSet`	The selection set of fields to include when this fragment is spread.
`SelectionSet`	`Node`	`selections`	`List[Selection]`	List of fields, fragment spreads, or inline fragments.
`Selection`	`Node`	(none, base class)		Abstract base for items in a selection set.
`Field`	`Selection`	`name`	`str`	The name of the field as defined in the schema (e.g., "email").
		`alias`	`Optional[str]`	Optional alias for the field (e.g., "userEmail: email").
		`arguments`	`List[Argument]`	List of arguments provided to this field.
		`directives`	`List[Directive]`	Field-level directives (e.g., `@include(if: $show)`).
		`selection_set`	`Optional[SelectionSet]`	Nested selection set for object-type fields. `None` for scalar fields.
`FragmentSpread`	`Selection`	`name`	`str`	Name of the fragment to spread (e.g., "UserFields").

Type Name	Parent Type	Fields	Field Type	Description
		`directives`	`List[Directive]`	Directives applied to this spread.
`InlineFragment`	`Selection`	`type_condition`	`Optional[NamedType]`	Optional type condition (e.g., `... on User`). If `None`, it's an unconditional inline fragment.
		`directives`	`List[Directive]`	Directives applied to this inline fragment.
		`selection_set`	`SelectionSet`	The selection set for this fragment.
`Argument`	`Node`	`name`	`str`	Name of the argument (e.g., "id").
		`value`	`Value`	The value provided, which can be a literal, variable, list, or object.
`Directive`	`Node`	`name`	`str`	Name of the directive (e.g., "include").
		`arguments`	`List[Argument]`	Arguments passed to the directive.
`VariableDefinition`	`Node`	`variable`	`Variable`	The variable being defined (e.g., `$id`).
		`type`	`Type`	The expected GraphQL type (e.g., `ID!`).
		`default_value`	`Optional[Value]`	Optional default value if the variable is not provided.
`Variable`	`Value`	`name`	`str`	Name of the variable without the `$` prefix (e.g., "id").
`NamedType`	`Type`	`name`	`str`	Name of the referenced type (e.g., "User", "ID").
`ListType`	`Type`	`type`	`Type`	The type of items in the list (e.g., `[String]`).
`NonNullType`	`Type`	`type`	`Type`	The wrapped type that is made non-null (e.g., `String!`).
`Value`	`Node`	(none, base class)		Abstract base for all value literals and variables.
`IntValue`	`Value`	`value`	`int`	Integer literal (e.g., `42`).
`FloatValue`	`Value`	`value`	`float`	Floating-point literal (e.g., `3.14`).
`StringValue`	`Value`	`value`	`str`	String literal (e.g., `"hello"`).
`BooleanValue`	`Value`	`value`	`bool`	Boolean literal (`true` or `false`).
`EnumValue`	`Value`	`value`	`str`	Enum literal (e.g., `PUBLISHED`).
`NullValue`	`Value`	(none)		Represents the literal `null`.
`ListValue`	`Value`	`values`	`List[Value]`	List of values (e.g., `["a", "b"]`).
`ObjectValue`	`Value`	`fields`	`List[ObjectField]`	Key-value pairs (e.g., `{x: 1, y: 2}`).
`ObjectField`	`Node`	`name`	`str`	Field name in the object literal.
		`value`	`Value`	The value for this field.

> **Architecture Decision: Immutable AST**
>
> - **Context**: The AST is produced by the parser and then read by multiple downstream components (validator, executor, compiler). These components may traverse the tree concurrently or in multiple passes.
> - **Options Considered**:
>   1. **Mutable AST with parent pointers**: Nodes contain references to their parent, allowing easy upward traversal. Nodes can be modified in-place (e.g., fragment expansion).
>   2. **Immutable AST without parent pointers**: Nodes are frozen after creation; traversal is only downward via children. Modifications require creating new trees.
> - **Decision**: Immutable AST without parent pointers.
> - **Rationale**:
>   - **Thread Safety**: Immutability guarantees safe concurrent traversal, which is useful for parallel field execution and static analysis.
>   - **Predictability**: The AST reflects exactly what was parsed, making debugging easier. Fragment expansion can be done as a separate transformation pass that produces a new tree.
>   - **Simpler Memory Management**: No circular references (parent-child) ease garbage collection in some languages and prevent accidental leaks.
> - **Consequences**:
>   - Fragment expansion and other transformations must copy the affected subtree, which has a memory/performance cost.
>   - Traversing upward (e.g., from a field to its enclosing operation) requires external context tracking or a separate visitor pattern that maintains a stack.

Common Pitfalls in AST Design:

- ⚠️ **Pitfall: Forgetting Location Tracking**
  - **Description**: Omitting the `loc` field or not populating it during parsing.
  - **Why it's wrong**: Error messages become useless ("Error in query" without line/column). Debugging complex queries becomes a guessing game.
  - **Fix**: Ensure every node created by the parser receives a `Location` object. The tokenizer must track line/column numbers for each token.
- ⚠️ **Pitfall: Mixing Value and Type Hierarchies**
  - **Description**: Representing `IntValue` and `IntType` with the same class because both contain an integer.
  - **Why it's wrong**: They are fundamentally different concepts. A value is a runtime literal; a type is a schema definition. Merging them causes confusion in later stages (e.g., type checking).
  - **Fix**: Maintain separate class hierarchies for `Value` and `Type` nodes, even if they look similar.
- ⚠️ **Pitfall: Overlooking Directive Locations**
  - **Description**: Not storing directives on all applicable AST nodes (operations, fields, fragments, etc.).
  - **Why it's wrong**: Directives like `@include`, `@skip`, or custom directives are part of the query semantics and must be available to the executor.
  - **Fix**: Include a `directives: List[Directive]` field on every node type that can have directives per the GraphQL spec.

Type System Types

Mental Model: The Building Code If the GraphQL schema were a city, the type system would be its building code. It defines what structures are allowed (`ObjectType` = houses, `ScalarType` = bricks), how they connect (`InterfaceType` = blueprints for neighborhoods), and the rules they must follow (`NonNull` = required safety features). Just as a building inspector ensures constructions follow the code, the type system validates that queries and data conform to the schema.

The type system represents the GraphQL schema defined by the user or reflected from the database (Milestone 2, Milestone 4). It is a runtime model used to validate queries, guide execution, and generate introspection responses. Each GraphQL type kind has a corresponding class, with a common base `GraphQLType`.

Type System Class Hierarchy and Data Structures

The following table details all type system classes, their fields, and purposes. The hierarchy is: `GraphQLType` → `ScalarType`, `ObjectType`, `InterfaceType`, `UnionType`, `EnumType`, `InputObjectType`, with wrapper types `ListType` and `NonNullType` also extending

`GraphQLType`.

Type Name	Parent Type	Fields	Field Type	Description
`GraphQLType`	(none)	`name`	`str`	The name of the type (e.g., "User", "ID"). For wrapper types, this is often derived (e.g., "[User]").
		`description`	`Optional[str]`	Optional description for documentation/introspection.
`ScalarType`	`GraphQLType`	`serialize`	`Callable[[Any], Any]`	Function to convert a Python value to a suitable output format (e.g., datetime to ISO string).
		`parse_value`	`Callable[[Any], Any]`	Function to convert a variable value (JSON) to the internal representation.
		`parse_literal`	`Callable[[Value], Any]`	Function to convert an AST literal node to the internal representation.
`ObjectType`	`GraphQLType`	`fields`	`Dict[str, GraphQLField]`	Map of field name to field definition.
		`interfaces`	`List[InterfaceType]`	List of interfaces this object implements.
`GraphQLField`	(none)	`type`	`GraphQLType`	The return type of this field.
		`args`	`Dict[str, GraphQLArgument]`	Map of argument name to argument definition.
		`resolve`	`Optional[Callable]`	Optional resolver function. If `None`, a default property resolver is used.
		`description`	`Optional[str]`	Field description.
		`deprecation_reason`	`Optional[str]`	If set, the field is deprecated with this reason.
`GraphQLArgument`	(none)	`type`	`GraphQLType`	The expected type of the argument.
		`default_value`	`Any`	Default value if the argument is not provided.
		`description`	`Optional[str]`	Argument description.
`InterfaceType`	`GraphQLType`	`fields`	`Dict[str, GraphQLField]`	Map of field name to field definition (abstract).
		`resolve_type`	`Optional[Callable[[Any], ObjectType]]`	Optional function to determine the concrete object type for a given value.
`UnionType`	`GraphQLType`	`types`	`List[ObjectType]`	List of possible object types that this union can represent.
		`resolve_type`	`Optional[Callable[[Any], ObjectType]]`	Optional function to determine the concrete object type for a given value.
`EnumType`	`GraphQLType`	`values`	`Dict[str, EnumValue]`	Map of enum value name to its definition.
`EnumValue`	(none)	`value`	`Any`	The underlying value (often a string matching the name).
		`description`	`Optional[str]`	Description of this enum value.
		`deprecation_reason`	`Optional[str]`	Deprecation reason, if applicable.
`InputObjectType`	`GraphQLType`	`fields`	`Dict[str, GraphQLInputField]`	Map of field name to input field definition.

Type Name	Parent Type	Fields	Field Type	Description
`GraphQLInputField`	(none)	`type`	`GraphQLType`	The expected type of the input field.
		`default_value`	`Any`	Default value if not provided.
		`description`	`Optional[str]`	Field description.
`ListType`	`GraphQLType`	`of_type`	`GraphQLType`	The type of items in the list.
`NonNullType`	`GraphQLType`	`of_type`	`GraphQLType`	The wrapped type that is made non-null.
`Schema`	(none)	`query_type`	`ObjectType`	The root query object type.
		`mutation_type`	`Optional[ObjectType]`	Optional root mutation object type.
		`subscription_type`	`Optional[ObjectType]`	Optional root subscription object type.
		`types`	`Dict[str, GraphQLType]`	Map of all named types in the schema, keyed by name.
		`directives`	`Dict[str, GraphQLDirective]`	Map of all directives supported by the schema.
`GraphQLDirective`	(none)	`name`	`str`	Name of the directive (e.g., "include").
		`locations`	`List[str]`	Where this directive can be used (e.g., ["FIELD", "FRAGMENT_SPREAD"]).
		`args`	`Dict[str, GraphQLArgument]`	Arguments the directive accepts.
		`description`	`Optional[str]`	Directive description.

Architecture Decision: Dual-Type Representation for Wrappers

- **Context**: GraphQL has type modifiers: `List` and `NonNull`. These can be nested (e.g., `[String!]!`). We need to represent these in memory for type validation and introspection.
- **Options Considered**:
  1. **Inline flags**: Add `is_list` and `is_non_null` boolean fields to `GraphQLType`, with an `inner_type` field for the wrapped type. Nesting is represented by chaining `inner_type`.
  2. **Wrapper objects**: Create separate `ListType` and `NonNullType` classes that wrap another `GraphQLType`. Nesting is represented by a chain of wrapper objects.
- **Decision**: Wrapper objects (`ListType`, `NonNullType`).
- **Rationale**:
  - **Spec Compliance**: The GraphQL introspection system returns wrapper types as separate objects with `kind` fields. The wrapper object approach maps directly to this.
  - **Type Safety**: The type system can use isinstance checks (`isinstance(t, ListType)`) which is clearer than checking boolean flags.
  - **Recursive Simplicity**: Algorithms for unwrapping (e.g., getting the underlying named type) become simple loops (`while isinstance(t, WrapperType): t = t.of_type`).
- **Consequences**:
  - Slightly more memory overhead due to many small objects.
  - Equality checks require recursive traversal (e.g., two `[String!]` types are equal only if their inner types are equal).

Common Pitfalls in Type System Design:

- ⚠️ **Pitfall: Confusing Input and Output Types**
  - **Description**: Using the same class for `ObjectType` and `InputObjectType`, or allowing input types to be used as output field types.
  - **Why it's wrong**: GraphQL strictly separates input and output types. Input types cannot have arguments or interfaces; output types cannot be used as argument types. Mixing them leads to validation errors and runtime confusion.

    - **Fix**: Maintain separate class hierarchies for input and output types, with validation during schema building that ensures correct usage.
- ⚠️ **Pitfall: Missing Circular Reference Detection**
    - **Description**: Allowing a type to reference itself indefinitely (e.g., `type User { friends: [User] }` is valid, but `type Circular { field: Circular }` as an input type is not).
    - **Why it's wrong**: GraphQL forbids circular references in input types. Infinite recursion during validation or serialization can crash the engine.
    - **Fix**: During schema building, perform a depth-first search to detect cycles in input type references, and raise a validation error.
- ⚠️ **Pitfall: Forgetting Introspection Types**
    - **Description**: Not including the built-in introspection types (`__Schema` , `__Type` , `__Field` , etc.) in the schema's type map.
    - **Why it's wrong**: GraphQL introspection queries (used by tools like GraphiQL) will fail, breaking developer experience.
    - **Fix**: Always add the introspection types to the schema's `types` dictionary. They can be generated statically.

Execution Context and State

Mental Model: The Restaurant Ticket Imagine a GraphQL query as a customer's order at a restaurant. The **execution context** is the kitchen ticket attached to that order. It tracks the order's progress through different stations (resolvers), any special instructions (context variables like allergy info), errors ("86 salmon"), and the final assembled dishes (data). As the ticket moves through the kitchen, each station adds its output and notes.

The execution context holds all runtime information needed to resolve a GraphQL query (Milestone 3). It is passed to every resolver function, providing access to shared resources, the current field's information, and a place to collect errors. It also manages the **DataLoader** instances that batch and cache database requests to solve the N+1 problem.

Execution State Data Structures

Type Name	Fields	Field Type	Description
`ExecutionContext`	`schema`	`Schema`	The GraphQL schema being executed against.
	`document_ast`	`Document`	The parsed AST of the operation to execute.
	`operation_name`	`Optional[str]`	Name of the operation to execute (if multiple in document).
	`variable_values`	`Dict[str, Any]`	Map of variable name to coerced value.
	`context_value`	`Any`	The user-provided context object (often includes database connection, auth info).
	`root_value`	`Any`	The initial value for the root type (often `None`).
	`fragments`	`Dict[str, FragmentDefinition]`	Map of fragment name to AST definition, for quick lookup.
	`errors`	`List[GraphQLError]`	Errors collected during execution.
	`data_loaders`	`Dict[str, DataLoader]`	Map of loader name to DataLoader instance for batching.
	`path`	`List[PathSegment]`	Current execution path as a stack (e.g., `["user", "friends", 0, "name"]`).
	`field_resolver`	`Callable`	The default field resolver function (falls back to property access).
	`type_resolver`	`Callable`	The default type resolver for abstract types.
`GraphQLError`	`message`	`str`	Human-readable error message.
	`locations`	`List[Location]`	Source locations where the error occurred (from AST nodes).
	`path`	`List[PathSegment]`	Execution path to the field where the error originated.
	`original_error`	`Optional[Exception]`	The underlying exception that caused this error, if any.
	`extensions`	`Dict[str, Any]`	Optional map for custom error metadata.
`PathSegment`	(union type)	`str | int`	Either a field name (for object fields) or an index (for list items).
`DataLoader`	`batch_load_fn`	`Callable[[List[Any]], List[Any]]`	Function that receives a list of keys and returns a list of values or promises.
	`cache`	`Dict[Any, Any]`	Map of key to resolved value (or promise).
	`queue`	`List[Any]`	Keys pending batch resolution.
	`max_batch_size`	`int`	Maximum number of keys to batch in one call.
`ExecutionResult`	`data`	`Optional[Dict[str, Any]]`	The result data if execution succeeded (partial data may be present even with errors).
	`errors`	`List[GraphQLError]`	Errors encountered during execution.
	`extensions`	`Dict[str, Any]`	Optional extension metadata for the response.

Architecture Decision: Mutable Execution Context vs. Immutable Per-Resolve Context

- **Context**: Resolver functions need access to execution state (path, variables, context value). This state changes as execution traverses the query tree.
- **Options Considered**:
  1. **Mutable shared context**: A single `ExecutionContext` object passed to all resolvers, with mutable fields like `path` updated in-place as execution depth changes.
  2. **Immutable per-resolve context**: Create a new context object for each field resolution, copying or deriving state (e.g., appending to path).
- **Decision**: Mutable shared context with careful stack management.
- **Rationale**:
  - **Performance**: Creating new context objects for every field (potentially thousands) adds overhead. Mutation is cheaper.
  - **Simplicity**: The path can be managed as an explicit stack: push before resolving a field, pop after. This mirrors the natural recursion of execution.
  - **Predictability**: The context is single-threaded per request, so mutability is safe. Concurrency is handled at the field level via async, not threads.
- **Consequences**:
  - Resolvers must not modify context fields they shouldn't (e.g., they shouldn't mutate `path` directly). Provide helper methods (`context.push_path(), context.pop_path()`).
  - Special care needed for parallel execution of sibling fields—each parallel branch needs a snapshot of the path or must coordinate to avoid corrupting the shared path.

Algorithm for Field Execution with Context:

1. **Initialize context**: Create `ExecutionContext` with schema, document, variables, root value, and empty error list.
2. **Select operation**: If `operation_name` is provided, find the matching `OperationDefinition` in the document; otherwise, if there's exactly one operation, use it.
3. **Coerce variables**: For each `VariableDefinition` in the operation, coerce the input `variable_values` using the variable's type. Add errors to context for invalid values.
4. **Execute operation**: Call `execute_operation(context, root_type, root_value)` :
   1. Determine the root object type (`query_type` , `mutation_type` , or `subscription_type`).
   2. Create a `SelectionSet` from the operation's `selection_set` .
   3. Call `execute_fields(context, root_type, root_value, selection_set)` .
5. **Execute fields** (recursive):
   1. Create an empty `result_dict` .
   2. For each `selection` in the `selection_set` :
      - If selection is a `Field` :
        1. **Push path**: Append field's alias or name to `context.path` .
        2. **Resolve field**: Call `resolve_field(context, parent_type, parent_value, field)` .
        3. **Pop path**: Remove the last element from `context.path` .
        4. Store the resolved value in `result_dict` under the output key (alias or name).
      - If selection is a `FragmentSpread` or `InlineFragment` :
        1. Check type condition against `parent_type` . If it doesn't apply, skip.
        2. Recursively `execute_fields` with the fragment's `selection_set` .
        3. Merge results into `result_dict` .
   3. Return `result_dict` .
6. **Resolve field**:
   1. **Coerce arguments**: For each argument in the field's AST, coerce the value using the argument's type from the schema.
   2. **Get resolver**: Look up the resolver function for this field on the parent type. If none, use `context.field_resolver` .

3. **Call resolver**: Invoke resolver with `(parent_value, arguments, context, info)`, where `info` is a `ResolveInfo` object containing field details.
4. **Handle result**:
    - If resolver returns an exception, add a `GraphQLError` to context.
    - If resolver returns a value, coerce it to the field's declared type (handling nullability, lists, etc.).
    - If the coerced value is an object type and the field has a `selection_set`, recursively `execute_fields`.

Common Pitfalls in Execution State Design:

- ⚠️ **Pitfall: Not Propagating Null Correctly**
    - **Description**: When a non-null field resolves to `null`, the engine fails to bubble that null up to the nearest nullable parent, causing the entire query to fail incorrectly.
    - **Why it's wrong**: GraphQL spec requires that if a non-null field returns null, that null propagates, making the parent field null (or if the parent is also non-null, continuing up). This allows partial data to be returned with errors.
    - **Fix**: In `resolve_field`, after coercing the value, if the result is `null` and the field's type is `NonNullType`, add an error to context and return `null`. The caller (`execute_fields`) must detect this and propagate it.
- ⚠️ **Pitfall: Forgetting to Clone Path for Parallel Execution**
    - **Description**: When resolving sibling fields in parallel (e.g., `{ user { name email } }`), using a mutable shared `path` leads to race conditions where one field's path overwrites another's.
    - **Why it's wrong**: Errors will have incorrect paths, making debugging impossible. Execution may become non-deterministic.
    - **Fix**: Before dispatching parallel field resolution, take a snapshot of the current path (or create a new context copy) for each branch. Alternatively, make `path` immutable and pass it as an argument to `resolve_field`.
- ⚠️ **Pitfall: DataLoader Cache Pollution Across Requests**
    - **Description**: Using a single `DataLoader` instance across multiple GraphQL requests, causing cached data from one user to leak to another.
    - **Why it's wrong**: Security and data isolation violation. User A might see User B's data if IDs happen to match.
    - **Fix**: Create a new `DataLoader` instance per request, attached to `ExecutionContext.data_loaders`. The cache is then request-scoped and garbage-collected after the request.

SQL Intermediate Representation

Mental Model: The Construction Plan Translating a GraphQL query to SQL is like turning an architect's sketch (GraphQL) into a builder's detailed plan (SQL). The **SQL IR** is that plan—a step-by-step breakdown that says: "We need to query the `users` table, join with `posts` on `user_id`, filter where `status = 'active'`, and return columns `id`, `name`, and `posts.title`." It's an intermediate representation that's closer to SQL than GraphQL but not yet a specific SQL dialect string, allowing for optimizations and parameter binding.

The SQL IR bridges the GraphQL AST and the final SQL query (Milestone 5). It is a tree or pipeline structure that captures the relational operations needed to fulfill the GraphQL query: which tables to select from, how to join them, what filters to apply, and what columns to return. This IR can be optimized (e.g., reorder joins, push down filters) before being rendered to a specific SQL dialect.

SQL IR Data Structures

Type Name	Fields	Field Type	Description
`SQLQuery`	`root_select`	`SQLSelect`	The top-level SELECT statement (often corresponds to the GraphQL query root).
	`variables`	`Dict[str, Any]`	Map of parameter names to values for safe SQL parameterization.
	`result_mapper`	`Callable[[Dict], Dict]`	Function to transform flat SQL row results into nested GraphQL response shape.
`SQLSelect`	`from_table`	`SQLTable`	Primary table or subquery being selected from.
	`joins`	`List[SQLJoin]`	List of joins to other tables.
	`where`	`Optional[SQLExpression]`	WHERE clause condition expression.
	`order_by`	`List[SQLOrderBy]`	ORDER BY clauses.
	`limit`	`Optional[int]`	LIMIT clause.
	`offset`	`Optional[int]`	OFFSET clause.
	`columns`	`List[SQLColumn]`	Columns to select (can be from primary table or joined tables).
	`distinct`	`bool`	Whether to add DISTINCT.
	`group_by`	`List[SQLColumn]`	GROUP BY columns (for aggregations).
`SQLTable`	`name`	`str`	Database table name (or alias).
	`schema`	`Optional[str]`	Database schema (e.g., "public").
	`alias`	`Optional[str]`	Alias for this table in the query (e.g., "u" for "users AS u").
`SQLJoin`	`type`	`str`	Join type: "INNER", "LEFT", "RIGHT", "FULL".
	`table`	`SQLTable`	The table being joined.
	`on`	`SQLExpression`	ON clause condition (e.g., "users.id = posts.user_id").
`SQLColumn`	`table`	`SQLTable`	Table this column belongs to.
	`name`	`str`	Column name.
	`alias`	`Optional[str]`	Output alias (e.g., "userName" for "name AS userName").
	`expression`	`Optional[str]`	Optional raw SQL expression (e.g., "COUNT(*)" instead of a simple column).
`SQLExpression`	`kind`	`str`	Node type: "binary", "unary", "literal", "column", "parameter".
	`operator`	`Optional[str]`	For binary/unary: "=", ">", "AND", "NOT", etc.
	`left`	`Optional[SQLExpression]`	Left operand.
	`right`	`Optional[SQLExpression]`	Right operand.
	`value`	`Any`	For literal/parameter: the value.
	`column`	`Optional[SQLColumn]`	For column reference: the column.
`SQLOrderBy`	`column`	`SQLColumn`	Column to order by.
	`direction`	`str`	"ASC" or "DESC".
`SQLParameter`	`name`	`str`	Parameter placeholder name (e.g., "$1", ":id").
	`value`	`Any`	The value to bind.

Type Name	Fields	Field Type	Description
	`type`	`str`	SQL type hint (e.g., "int", "text").

> **Architecture Decision: SQL IR as a Tree vs. Pipeline**
>
> - **Context**: We need to represent relational operations that can be nested (subqueries, joins of joins). The structure should support both simple flat queries and complex nested ones.
> - **Options Considered**:
>   1. **Pipeline model**: A linear sequence of operations (Scan → Filter → Join → Project) similar to relational algebra. Each stage transforms a result set.
>   2. **Tree model**: A hierarchical SELECT tree where each SELECT can have JOIN children, and JOINs can have their own SELECT subtrees. This mirrors SQL's nested nature.
> - **Decision**: Tree model centered on `SQLSelect`.
> - **Rationale**:
>   - **Natural Mapping**: GraphQL's nested selections map directly to nested `SQLSelect` nodes (via joins). A field `user { posts { comments } }` becomes a main SELECT for users, a JOIN to posts, and a sub-SELECT for comments.
>   - **SQL Generation Simplicity**: Walking the tree and emitting SQL is straightforward—each `SQLSelect` renders to a SELECT clause, with joins rendered as JOIN clauses in the same SELECT.
>   - **Optimization Opportunities**: The tree can be rearranged (join reordering, filter push-down) by rewriting subtrees without changing the overall pipeline abstraction.
> - **Consequences**:
>   - Some operations that are naturally pipeline (e.g., LIMIT after ORDER BY) must be represented as fields within the same `SQLSelect` node, not as separate stages.
>   - Handling correlated subqueries (where a subquery references columns from outer query) requires careful design of column references across tree levels.

Algorithm for Generating SQL IR from GraphQL AST:

1. **Analyze selection set**: Traverse the GraphQL AST starting at the root field (e.g., `query { users { ... } }`). Identify the primary table mapping from the root field name (`users` → `users` table).
2. **Create root SQLSelect**: Instantiate `SQLSelect` with `from_table` set to the primary table.
3. **Process selections recursively**:
   - For each field in the selection set:
     - If field is a scalar (maps to a column):
       1. Add a `SQLColumn` to `columns` list, mapping GraphQL field name to table column name.
     - If field is an object (maps to a relationship):
       1. Determine relationship type (one-to-one, one-to-many) via foreign key metadata.
       2. For one-to-one: Add a `SQLJoin` to the related table, and recursively process the nested selection set, adding columns from the joined table.
       3. For one-to-many: This becomes either a **lateral join** (for nested array) or a separate batch query. For simplicity, initially implement as a separate batch query (DataLoader pattern).
4. **Apply filters**: Convert GraphQL field arguments (e.g., `where: { status: "active" }`) into `SQLExpression` trees attached to `where`.
5. **Apply pagination**: Convert `first`, `after`, `offset` arguments into `limit`/`offset` or cursor-based conditions.
6. **Apply ordering**: Convert `orderBy` argument into `SQLOrderBy` list.
7. **Collect parameters**: Extract all literal values from GraphQL arguments and replace them with `SQLParameter` placeholders in the SQL IR, storing the actual values in `SQLQuery.variables`.

Common Pitfalls in SQL IR Design:

- ⚠️ **Pitfall: Generating Cartesian Products**
  - **Description**: When joining multiple one-to-many relationships in the same query level (e.g., `user { posts, comments }`), a naive JOIN leads to a Cartesian product between posts and comments for each user.

- **Why it's wrong**: Results explode in size (user with 10 posts and 10 comments yields 100 rows), causing performance issues and incorrect data aggregation.
- **Fix**: For multiple one-to-many relationships, use **separate queries** (DataLoader batches) or **lateral joins** that keep the relationships separate. The SQL IR should mark such relationships as requiring separate batch execution.

- ⚠️ **Pitfall: Not Parameterizing Values**
  - **Description**: Embedding literal values directly into SQL expressions as strings, leading to SQL injection vulnerabilities.
  - **Why it's wrong**: Unsafe. A GraphQL argument `where: { name: "'; DROP TABLE users; --" }` could cause disaster.
  - **Fix**: Always use `SQLParameter` nodes for values. The SQL renderer should replace these with placeholders (`$1` , `%s`) and pass values separately to the database driver.

- ⚠️ **Pitfall: Ignoring Column Aliasing Conflicts**
  - **Description**: When joining tables with same column names (e.g., `users.id` and `posts.id`), selecting both without aliases leads to duplicate column names in result set, causing mapping errors.
  - **Why it's wrong**: Database drivers may overwrite one column with another, or the result mapper cannot distinguish which `id` belongs to which table.
  - **Fix**: Automatically alias columns in `SQLColumn` using a naming scheme (e.g., `users__id` , `posts__id`). The `result_mapper` must use these aliases to reconstruct nested objects.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option (for learning)	Advanced Option (for production)
AST & Type System	Pure Python classes with `dataclasses` or `attrs` for immutable nodes.	Use a library like `graphql-core` for reference implementation, or `pydantic` with validation.
Execution Context	Plain Python dict for context, manual path stack management.	Use `contextvars` for async context, `asyncio` for parallel resolution.
SQL IR	Simple namedtuple-like classes, string templating for SQL generation.	Use SQLAlchemy Core as a SQL abstraction layer, or a dedicated SQL builder like `pypika` .
DataLoader	Manual batching with dictionary caches per request.	Use `aiodataloader` library for async batching and caching.

Recommended File/Module Structure

```
graphql_engine/
├── ast/
│   ├── __init__.py
│   ├── nodes.py          # All AST node classes (Location, Document, Field, etc.)
│   └── parser.py         # Lexer and recursive descent parser (parse_query)
├── type_system/
│   ├── __init__.py
│   ├── types.py          # GraphQLType, ScalarType, ObjectType, etc.
│   ├── schema.py         # Schema class and validation logic
│   └── introspection.py  # Introspection type definitions and resolver
├── execution/
│   ├── __init__.py
│   ├── context.py        # ExecutionContext, GraphQLError
│   ├── executor.py       # execute_query, field resolution algorithms
│   ├── resolver.py       # Default field resolver, type resolver
│   └── dataloader.py     # DataLoader class
├── sql/
│   ├── __init__.py
│   ├── ir.py             # SQL IR classes (SQLQuery, SQLSelect, etc.)
│   ├── compiler.py       # AST-to-SQL IR compilation logic
│   └── renderer.py       # SQL IR to parameterized SQL string (for PostgreSQL, SQLite)
├── reflection/
│   ├── __init__.py
│   ├── introspector.py   # Database metadata queries
│   └── mapper.py         # Maps tables to GraphQL types, columns to fields
└── server.py             # HTTP server (optional, for testing)
```

Infrastructure Starter Code

Here is a complete, ready-to-use implementation of the `Location` and base `Node` classes, which are prerequisites for the AST:

```python
# ast/nodes.py

from typing import Optional, List, Union, Any

from dataclasses import dataclass


@dataclass(frozen=True)

class Location:

    """Source location in a GraphQL document."""

    line: int      # 1-indexed

    column: int    # 1-indexed


    def __str__(self) -> str:

        return f"{self.line}:{self.column}"


class Node:

    """Base class for all AST nodes."""

    __slots__ = ('loc',)  # Saves memory, ensures immutability


    def __init__(self, loc: Optional[Location] = None):

        self.loc = loc


    def __repr__(self) -> str:

        cls_name = self.__class__.__name__

        fields = ", ".join(f"{k}={v!r}" for k, v in self.__dict__.items() if k != 'loc')

        if self.loc:

            return f"{cls_name}({fields}, loc={self.loc})"

        return f"{cls_name}({fields})"
```

Core Logic Skeleton Code

AST Node Skeleton (to be completed by learner):

```python
# ast/nodes.py (continued)

@dataclass(frozen=True)
class Document(Node):
    """Complete GraphQL document with definitions."""
    definitions: List['Definition']

    # TODO: Add method to get operation by name
    # TODO: Add method to get fragment by name

class Definition(Node):
    """Base for OperationDefinition and FragmentDefinition."""
    pass

@dataclass(frozen=True)
class OperationDefinition(Definition):
    """A query, mutation, or subscription operation."""
    operation_type: str  # 'query', 'mutation', 'subscription'
    name: Optional[str]
    variable_definitions: List['VariableDefinition']
    directives: List['Directive']
    selection_set: 'SelectionSet'

    # TODO: Add method to validate variable usage
    # TODO: Add method to collect referenced fragment names

@dataclass(frozen=True)
class SelectionSet(Node):
    """A set of fields/fragments selected on an object."""
    selections: List['Selection']

    # TODO: Add method to flatten fragments (expand FragmentSpread)
    # TODO: Add method to get all field names

# TODO: Define remaining AST classes: Field, FragmentDefinition, FragmentSpread,
#       InlineFragment, Argument, Directive, VariableDefinition, Variable,
#       NamedType, ListType, NonNullType, and all Value subclasses (IntValue, etc.)
#       Follow the table structure exactly.
```

Type System Skeleton:

PYTHON

```python
# type_system/types.py

from typing import Dict, List, Optional, Callable, Any


class GraphQLType:

    """Base class for all GraphQL types."""

    def __init__(self, name: str, description: Optional[str] = None):

        self.name = name

        self.description = description


    # TODO: Add method to get introspection representation (__Type)

    # TODO: Add method to check type equality (handling wrappers)


class ScalarType(GraphQLType):

    """GraphQL scalar type."""

    def __init__(self, name: str, serialize: Callable, parse_value: Callable,

                 parse_literal: Callable, description: Optional[str] = None):

        super().__init__(name, description)

        self.serialize = serialize

        self.parse_value = parse_value

        self.parse_literal = parse_literal


    # TODO: Add built-in scalars (Int, Float, String, Boolean, ID) as class constants


class ObjectType(GraphQLType):

    """GraphQL object type."""

    def __init__(self, name: str, fields: Dict[str, 'GraphQLField'],

                 interfaces: Optional[List['InterfaceType']] = None,

                 description: Optional[str] = None):

        super().__init__(name, description)

        self.fields = fields

        self.interfaces = interfaces or []


    # TODO: Add method to get field by name (including from interfaces)

    # TODO: Add method to validate interface implementations


# TODO: Define GraphQLField, GraphQLArgument, InterfaceType, UnionType,

#       EnumType, EnumValue, InputObjectType, GraphQLInputField,

#       ListType, NonNullType, Schema, GraphQLDirective.

#       Follow the table structure exactly.
```

ExecutionContext Skeleton:

PYTHON

```python
# execution/context.py

from typing import Dict, List, Optional, Any, Union

from dataclasses import dataclass, field

from ..ast.nodes import Document, Location

from ..type_system.schema import Schema


PathSegment = Union[str, int]


@dataclass

class ExecutionContext:

    """Runtime context for executing a GraphQL query."""

    schema: Schema

    document_ast: Document

    operation_name: Optional[str] = None

    variable_values: Dict[str, Any] = field(default_factory=dict)

    context_value: Any = None

    root_value: Any = None

    fragments: Dict[str, Any] = field(default_factory=dict)  # Maps name to FragmentDefinition

    errors: List['GraphQLError'] = field(default_factory=list)

    data_loaders: Dict[str, 'DataLoader'] = field(default_factory=dict)

    path: List[PathSegment] = field(default_factory=list)

    field_resolver: Callable = None

    type_resolver: Callable = None


    # TODO: Add method push_path(segment) -> context manager (for `with context.push_path(...):`)

    # TODO: Add method get_data_loader(key, batch_load_fn) -> returns existing or creates new DataLoader

    # TODO: Add method to coerce variable values based on operation's variable definitions


@dataclass

class GraphQLError(Exception):

    """GraphQL execution error with location and path."""

    message: str

    locations: List[Location] = field(default_factory=list)

    path: List[PathSegment] = field(default_factory=list)

    original_error: Optional[Exception] = None

    extensions: Dict[str, Any] = field(default_factory=dict)


    # TODO: Add method to format error for JSON response (per GraphQL spec)

    # TODO: Add class method from_exception to wrap Python exceptions
```

```
# TODO: Implement DataLoader class with batch scheduling and caching.
```

SQL IR Skeleton:

```python
# sql/ir.py

from typing import List, Optional, Dict, Any, Callable

from dataclasses import dataclass


@dataclass

class SQLQuery:

    """Complete SQL query representation."""

    root_select: 'SQLSelect'

    variables: Dict[str, Any] = field(default_factory=dict)

    result_mapper: Optional[Callable[[Dict], Dict]] = None


    # TODO: Add method to render to SQL string with placeholders

    # TODO: Add method to execute via database adapter


@dataclass

class SQLSelect:

    """A SELECT statement."""

    from_table: 'SQLTable'

    joins: List['SQLJoin'] = field(default_factory=list)

    where: Optional['SQLExpression'] = None

    order_by: List['SQLOrderBy'] = field(default_factory=list)

    limit: Optional[int] = None

    offset: Optional[int] = None

    columns: List['SQLColumn'] = field(default_factory=list)

    distinct: bool = False

    group_by: List['SQLColumn'] = field(default_factory=list)


    # TODO: Add method to add column with auto-alias

    # TODO: Add method to find join by table name


# TODO: Define SQLTable, SQLJoin, SQLColumn, SQLExpression, SQLOrderBy, SQLParameter.

#       Follow the table structure exactly.
```

Language-Specific Hints (Python)

- Use `@dataclass(frozen=True)` for immutable AST nodes to ensure hashability and prevent accidental mutation.
- For type checking, use `typing` module extensively. Consider `Protocol` for resolver interfaces.
- Use `contextvars.ContextVar` for async context if you implement async resolvers.

- For SQL parameterization, use `%s` placeholders with PostgreSQL's `psycopg2` or SQLite's `sqlite3`. Never use string formatting.
- Use `asyncio.gather` for parallel field resolution in async executors.
- Consider `functools.lru_cache` for memoizing schema introspection results.

Milestone Checkpoint for Data Model

After implementing the data structures above (but before implementing parsers or executors), you should be able to run:

```bash
python -m pytest graphql_engine/ast/nodes.py -v
```

Expected: Tests for AST node creation and equality should pass. Example test:

```python
def test_field_node():
    loc = Location(line=1, column=2)
    field = Field(name="id", alias=None, arguments=[], directives=[], selection_set=None, loc=loc)
    assert field.name == "id"
    assert field.loc == loc
    assert isinstance(field, Node)
```

Signs something is wrong:

- **Error**: `TypeError: cannot set attribute` on a frozen dataclass. → You tried to modify an immutable node. Ensure you're not mutating after creation.
- **Error**: `RecursionError` when printing a node. → Likely circular reference in `__repr__`. Check that you're not following relationships indefinitely (e.g., `Field` referencing its parent `SelectionSet`).
- **Warning**: Mypy reports missing fields. → Check that you've implemented all fields from the tables above.

Component 1: GraphQL Parser

> **Milestone(s):** This section corresponds to Milestone 1: GraphQL Parser, which focuses on converting GraphQL query strings into Abstract Syntax Trees (ASTs).

Responsibility and Scope

The GraphQL Parser is the system's **front door**—it takes raw GraphQL query strings from clients and transforms them into structured, machine-readable representations. Think of it as the **optical character recognition** system for a document scanner: it doesn't understand the content's meaning (that's for validators and executors), but it must perfectly recognize and digitize every character, word, and sentence structure.

Responsibility	In Scope	Out of Scope
Lexical Analysis	Break query string into tokens (keywords, identifiers, punctuation)	Validate token semantics (e.g., "query" vs. "mutation" meaning)
Syntactic Analysis	Build AST following GraphQL grammar rules	Validate AST against schema (type checking, field existence)
Source Location Tracking	Record line/column for every AST node	Error recovery and intelligent suggestions
Document Structure	Parse operations, fragments, variables, directives	Execute operations or resolve field values
Literal Value Parsing	Convert string/number/boolean literals to Python values	Validate literal ranges (e.g., Int within 32-bit range)
Query Document Assembly	Build complete `Document` AST with all definitions	Optimize or normalize the query structure

The parser's output is a **syntactically correct but semantically unvalidated** AST. It ensures the query is well-formed according to GraphQL's grammar, but doesn't verify that referenced types exist, fields are valid, or variables match their declared types. This separation follows the **compiler principle** of separating syntax (form) from semantics (meaning).

Mental Model: Language Translator

Imagine you're **translating a restaurant order from a customer's scribbled note into a structured kitchen ticket**. The customer writes:

```
Large pepperoni pizza, extra cheese
No mushrooms
Add: 2 garlic bread
Diet Coke
```

A human translator would:

1. **Tokenize**: Recognize "Large" as size, "pepperoni" as topping, "2" as quantity
2. **Structure**: Group items into categories (pizza, sides, drinks)
3. **Resolve references**: Link "extra cheese" to the pizza, not the garlic bread
4. **Handle modifiers**: Apply "No mushrooms" as an exclusion, not an addition

The GraphQL parser does exactly this translation:

- **Tokens** = individual words and punctuation (`{` , `query` , `user` , `name`)
- **Structure** = nested selection sets with fields and arguments
- **References** = fragment spreads pointing to their definitions
- **Modifiers** = directives (`@include` , `@skip`) and variable definitions

The key insight is that the parser **doesn't check if pepperoni is on the menu** (schema validation) or **whether the kitchen has garlic bread** (execution)—it only ensures the order is written in valid restaurant-language syntax.

Parser Interface

The parser exposes a minimal public API with two primary methods, following the principle of **single responsibility**—each method handles one specific input format.

Method	Parameters	Returns	Description	Side Effects
`parse_query`	`query_str: str`	`Document`	Parse GraphQL query/mutation/subscription	None (pure function)
`parse_schema`	`schema_str: str`	`List[TypeDefinition]`	Parse GraphQL Schema Definition Language	None (pure function)

Error Contract: Both methods raise `GraphQLSyntaxError` (a subclass of `GraphQLError`) for malformed input. The error includes:

- `message` : Human-readable description
- `locations` : List of `Location` objects (line, column)
- `original_error` : The underlying parsing exception if applicable

Input Examples:

```graphql
# parse_query input
query GetUser($id: ID!) {
  user(id: $id) {
    name
    email
  }
}

# parse_schema input
type User {
  id: ID!
  name: String
  email: String
}
```

Output Structure: The `parse_query` method returns a `Document` containing:

- `definitions`: List of operation and fragment definitions
- Source location information on every node
- Complete tree structure mirroring the query's nesting

Parsing Algorithm

The parser implements **recursive descent parsing with single-token lookahead**, which naturally matches GraphQL's nested but not overly complex grammar. The algorithm proceeds in two phases: **lexical analysis** (tokenization) followed by **syntactic analysis** (AST construction).

Phase 1: Lexical Analysis (Tokenizer)

The tokenizer scans the query string left-to-right, grouping characters into tokens while discarding meaningless whitespace and comments.

Token Types Table:

Token Type	Examples	Description	Special Handling
`NAME`	`query`, `user`, `_id`	GraphQL names (letters, digits, underscores)	Case-sensitive, cannot start with digit
`INT`	`42`, `0`, `-10`	Integer values	No decimal point, supports negative
`FLOAT`	`3.14`, `-2.5e-3`	Floating-point values	Scientific notation support
`STRING`	`"hello"`, `"line\nbreak"`	String literals	Escapes: `\\`, `\"`, `\/`, `\b`, `\f`, `\n`, `\r`, `\t`, `\uXXXX`
`PUNCTUATION`	`{`, `}`, `(`, `)`, `:`, `@`, `$`, `!`	Syntax punctuation	Single characters
`SPREAD`	`...`	Fragment spread operator	Three consecutive periods
`BLOCK_STRING`	`"""multiline string"""`	Triple-quoted strings	Handles leading/trailing whitespace specially

Tokenizer Algorithm:

1. **Initialize**: Set `position = 0`, `line = 1`, `column = 1`
2. **Skip ignorables**: Advance past whitespace (, `\t`, `,`) and comments (`# comment`)
3. **Identify token**:
   - If `char == '"'`: Check for `"""` (block string) else parse regular string
   - If `char.isalpha() or char == '_'`: Parse NAME token
   - If `char.isdigit() or (char == '-' and next_char.isdigit())`: Parse INT/FLOAT
   - If `char == '.'`: Check for `...` (SPREAD) else error
   - Otherwise: Match single-character PUNCTUATION
4. **Emit token**: Create token with value, line, column, and advance position
5. **Repeat** until end of input

Location Tracking: Each token records its starting `(line, column)` position, enabling precise error reporting. The tokenizer increments:

- `line` on `\n` or `\r\n`
- `column` resets to 1 after newline, increments by 1 otherwise

Phase 2: Syntactic Analysis (Parser)

The parser consumes tokens using **predictive parsing**—at each step, it examines the next token to decide which grammar rule applies. GraphQL's grammar is LL(1), meaning one token lookahead is sufficient.

Grammar Rules Simplified:

```
Document → Definition+
Definition → OperationDefinition | FragmentDefinition
OperationDefinition → OperationType Name? VariableDefinitions? Directives? SelectionSet
SelectionSet → '{' Selection+ '}'
Selection → Field | FragmentSpread | InlineFragment
Field → Alias? Name Arguments? Directives? SelectionSet?
```

Parsing Algorithm Steps:

1. **Parse Document**:
   - While tokens remain, parse a Definition
   - Add to `definitions` list
   - Return `Document(definitions)`
2. **Parse Definition**:
   - Peek at next token:
     - `NAME` = could be operation or fragment
     - `{` = anonymous operation
     - Otherwise error
   - If token is `fragment` : Parse `FragmentDefinition`
   - Else: Parse `OperationDefinition`
3. **Parse OperationDefinition**:
   - Parse operation type (`query` , `mutation` , `subscription`)
   - Optional: Parse name (if `NAME` token)
   - Optional: Parse variable definitions (if `(` token)
   - Optional: Parse directives
   - Parse `SelectionSet`
4. **Parse SelectionSet** (recursive core):
   - Expect `{` token
   - While next token ≠ `}` : Parse Selection
   - Expect `}` token
   - Return `SelectionSet(selections)`
5. **Parse Selection**:
   - Peek at tokens:
     - `...` → `FragmentSpread` or `InlineFragment`
     - `NAME` → `Field`
   - For `Field` : Parse optional alias, name, arguments, directives, optional nested `SelectionSet`
6. **Parse Fragments**:
   - `FragmentSpread` : Parse `...` followed by name and optional directives
   - `InlineFragment` : Parse `...` optional type condition, directives, and `SelectionSet`
   - `FragmentDefinition` : Parse `fragment` keyword, name, type condition, directives, `SelectionSet`
7. **Parse Value Literals** (used in arguments, defaults):
   - Dispatch based on token type:
     - `$` → `Variable`
     - `NAME` → `EnumValue` or `BooleanValue` or `NullValue`
     - `[` → `ListValue`
     - `{` → `ObjectValue`
     - Number/string tokens → corresponding scalar values

Recursive Nature: Notice how `parseSelectionSet` calls `parseSelection`, which for `Field` may call `parseSelectionSet` again—this **recursive descent** naturally handles GraphQL's arbitrarily nested structure.

Example Walkthrough: Parsing `query { user { name } }`

1. Token stream: [`NAME(query)`, `{`, `NAME(user)`, `{`, `NAME(name)`, `}`, `}`]
2. `parseDocument`: Parse Definition
3. `parseDefinition`: Anonymous operation (starts with `{`)
4. `parseOperationDefinition`: Default operation type = "query", parse `SelectionSet`
5. `parseSelectionSet`: Expect `{`, parse Selection(s)
6. `parseSelection`: `Field` with name "user", parse nested `SelectionSet`
7. `parseSelectionSet`: Expect `{`, parse Selection(s)
8. `parseSelection`: `Field` with name "name", no nested selection set
9. Backtrack, building nested AST: `Field(name="name")` → `SelectionSet([field])` → `Field(name="user", selection_set=SelectionSet)` → `SelectionSet([field])` → `OperationDefinition` → `Document`

Architecture Decision: Parser Approach

> **Decision: Recursive Descent Parser Over Parser Generator**
>
> **Context**: GraphQL has a well-specified, moderately complex grammar with nested structures (selection sets within fields), optional elements (aliases, directives), and context-sensitive rules (fragments must be defined before use in single document). We need a parser that is educational, debuggable, and maintainable for learners.
>
> **Options Considered**:
>
> 1. **Parser Generator (ANTLR, Lark)**: Define grammar in BNF/EBNF, generate parser code
> 2. **Parsing Expression Grammar (PEG)**: Use PEG library with grammar rules
> 3. **Manual Recursive Descent**: Hand-written parser with explicit parsing methods
>
> **Decision**: Implement manual recursive descent parser in pure Python.
>
> **Rationale**:
>
> - **Educational Value**: Recursive descent makes the parsing process transparent—learners can trace execution step-by-step, seeing exactly how tokens become AST nodes. Parser generators are "magic boxes" that hide the transformation.
> - **Error Reporting Quality**: Hand-written parsers can produce precise, context-aware error messages ("Expected '}' after selection set" vs. generic "syntax error").
> - **Incremental Development**: We can implement and test one grammar rule at a time (parseField, parseArgument, etc.).
> - **No External Dependencies**: Keeps the project self-contained and avoids version compatibility issues.
> - **GraphQL-Specific Optimizations**: We can add memoization for repeated fragments or optimize based on GraphQL's specific patterns.
>
> **Consequences**:
>
> - **More Initial Code**: ~500-800 lines vs. ~50 lines of grammar definition
> - **Maintenance Burden**: Grammar changes require manual code updates
> - **Potential for Bugs**: More opportunities for off-by-one errors in lookahead logic
> - **Excellent Debuggability**: Stack traces show exactly which parsing method failed

Comparison Table:

Option	Pros	Cons	Why Not Chosen
Parser Generator	Concise grammar definition, automatic error recovery, proven correctness	Opaque transformation, steep learning curve, external dependency	Hides the learning value—students wouldn't understand how parsing works
PEG Library	Natural grammar expression, good error messages, Python integration	Still hides implementation details, performance overhead	Less educational than seeing the recursive calls explicitly
Recursive Descent	Transparent, excellent error messages, no dependencies, debuggable	More code to write, manual maintenance	**CHOSEN**: Maximizes learning, aligns with compiler education tradition

Implementation Note: We use **single-token lookahead** rather than backtracking because GraphQL's grammar is LL(1). The parser examines the next token (`peek()`) to decide which path to take, without needing to rewind. This keeps the implementation simple and efficient.

Common Parser Pitfalls

⚠️ Pitfall: Incorrect String Escaping

- **Description**: Treating `\"` as two characters (backslash + quote) rather than an escaped quote
- **Why Wrong**: Results in malformed string values, breaks JSON serialization
- **Fix**: Implement proper escape sequence handling: `\\` , `\"` , `\/` , `\b` , `\f` , `\n` , `\r` , `\t` , `\uXXXX` (4 hex digits)
- **Test Case**: `"Quote: \", Newline: \n, Unicode: \u00A9"` → Should parse as `Quote: ", Newline: (newline), Unicode: ©`

⚠️ Pitfall: Missing Block String Handling

- **Description**: Treating `"""multiline\nstring"""` as three separate string tokens
- **Why Wrong**: Block strings have different escaping rules and handle whitespace specially per GraphQL spec
- **Fix**: Detect `"""` sequence, parse until closing `"""` , apply "trim indentation" algorithm
- **Test Case**: `"""Hello\n World"""` → Should parse as `Hello\n World` (no trimming since consistent indent)

⚠️ Pitfall: Location Tracking Offsets

- **Description**: Line/column counters get misaligned after multi-character tokens or newlines
- **Why Wrong**: Error messages point to wrong location, confusing debugging
- **Fix**: Update position counters **before** emitting token (token gets starting position), handle `\r\n` as single newline
- **Diagnostic**: Parse `{\n user\n}` → Field "user" should be at line 2, column 3, not line 1, column 5

⚠️ Pitfall: Fragment Spread vs. Inline Fragment Confusion

- **Description**: Misparsing `... on User` as a fragment spread named "on" followed by "User"
- **Why Wrong**: `on` is a keyword in type conditions, not a valid fragment name
- **Fix**: After `...` , check if next token is `NAME` and **not** `on` → fragment spread; if `on` → inline fragment
- **Edge Case**: `... on` (no type condition) is invalid but should be caught in validation, not parsing

⚠️ Pitfall: Variable vs. Argument Ambiguity

- **Description**: Confusing `field(arg: $value)` (argument with variable value) vs. `$value` alone (variable definition)
- **Why Wrong**: Parser tries to parse `$value` as an argument name (missing colon)
- **Fix**: `$` is only valid starting variable tokens; in argument context, `:` must follow argument name before value
- **Rule**: In `parseArguments` , after parsing name, expect `:` , then parse value (which may be a `Variable`)

⚠️ Pitfall: Unicode in Names

- **Description**: Rejecting valid Unicode characters in GraphQL names like `user_123` or `_valid`
- **Why Wrong**: GraphQL names support `[_A-Za-z][_0-9A-Za-z]*` with Unicode letters allowed
- **Fix**: Use Unicode-aware character classification: `char.isalpha()` or `char == '_'` for first char
- **Test**: `query_αβγ { field }` should parse correctly (Greek letters in name)

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option	Recommendation
Tokenizer	Manual character-by-character scanning	Regex-based with `re.Scanner`	**Manual scanning** (educational, precise control)
Token Representation	`NamedTuple` with type/value/location	Dataclass with methods	**`NamedTuple`** (immutable, lightweight, clear)
Parser State	Instance variables on Parser class	Functional style with returned state tuples	**Instance variables** (simpler to debug)
Error Handling	Exception-based with custom `GraphQLSyntaxError`	Result type (`Ok` / `Err`) monadic style	**Exceptions** (Pythonic, easy control flow)
AST Nodes	Dataclasses with inheritance	`NamedTuple` with manual dispatch	**Dataclasses** (clean, mutable during building)

B. Recommended File/Module Structure

```
graphql-engine/
├── pyproject.toml              # Project configuration
├── README.md
├── src/
│   └── graphql_engine/
│       ├── __init__.py
│       ├── error.py            # GraphQLError, GraphQLSyntaxError
│       ├── ast/                # AST node definitions
│       │   ├── __init__.py
│       │   ├── nodes.py        # All AST node classes
│       │   └── location.py     # Location class
│       ├── parser/
│       │   ├── __init__.py     # Public API: parse_query, parse_schema
│       │   ├── lexer.py        # Token, Tokenizer, TokenType enum
│       │   ├── parser.py       # Parser class with recursive methods
│       │   └── grammar.py      # Grammar constants (keywords, punctuation)
│       └── utils/
│           └── helpers.py      # String helpers, position tracking
└── tests/
    ├── __init__.py
    ├── test_lexer.py
    ├── test_parser.py
    └── fixtures/
        └── example_queries.graphql
```

C. Infrastructure Starter Code

`src/graphql_engine/error.py` (Complete implementation):

```python
"""GraphQL error definitions."""

from dataclasses import dataclass, field
from typing import List, Optional, Any, Union


@dataclass
class Location:
    """Source location for error reporting."""
    line: int
    column: int

    def __str__(self) -> str:
        return f"{self.line}:{self.column}"


@dataclass
class GraphQLError(Exception):
    """Base class for all GraphQL errors."""
    message: str
    locations: List[Location] = field(default_factory=list)
    path: List[Union[str, int]] = field(default_factory=list)
    original_error: Optional[Exception] = None
    extensions: dict[str, Any] = field(default_factory=dict)

    def __str__(self) -> str:
        if self.locations:
            locs = ", ".join(str(loc) for loc in self.locations)
            return f"{self.message} (locations: {locs})"
        return self.message


@dataclass
class GraphQLSyntaxError(GraphQLError):
    """Syntax error during parsing."""

    @classmethod
    def from_lexer(cls, message: str, line: int, column: int) -> "GraphQLSyntaxError":
        """Create syntax error from lexer position."""
        return cls(
```

```
            message=f"Syntax error: {message}",
            locations=[Location(line=line, column=column)]
        )

    @classmethod
    def from_parser(cls, message: str, token) -> "GraphQLSyntaxError":
        """Create syntax error from parser token."""
        return cls(
            message=f"Syntax error: {message} at '{token.value}'",
            locations=[Location(line=token.line, column=token.column)]
        )
```

`src/graphql_engine/ast/location.py` (Complete implementation):

```python
"""AST location tracking."""

from dataclasses import dataclass
from typing import Optional


@dataclass
class Location:
    """Source location for AST nodes."""
    line: int
    column: int

    @classmethod
    def from_token(cls, token) -> Optional["Location"]:
        """Create Location from token if token has location info."""
        if hasattr(token, 'line') and hasattr(token, 'column'):
            return cls(line=token.line, column=token.column)
        return None
```

`src/graphql_engine/utils/helpers.py` (Complete implementation):

```python
"""String and position utilities."""

import re
from typing import Tuple


def is_name_start(char: str) -> bool:
    """Check if character can start a GraphQL name."""
    return char.isalpha() or char == '_'


def is_name_continuation(char: str) -> bool:
    """Check if character can continue a GraphQL name."""
    return char.isalnum() or char == '_'


def decode_unicode_escape(match: re.Match) -> str:
    """Decode \uXXXX escape sequence."""
    hex_str = match.group(1)
    try:
        return chr(int(hex_str, 16))
    except ValueError:
        # Invalid Unicode, return replacement character
        return '\ufffd'


def parse_string_value(string_literal: str) -> str:
    """Parse GraphQL string literal with escape sequences."""
    # Remove surrounding quotes
    content = string_literal[1:-1]

    # Replace escape sequences
    replacements = [
        (r'\\"', '"'),
        (r'\\\\', '\\'),
        (r'\\/', '/'),
        (r'\\b', '\b'),
        (r'\\f', '\f'),
        (r'\\n', '\n'),
        (r'\\r', '\r'),
        (r'\\t', '\t'),
```

```
    ]

    for pattern, replacement in replacements:
        content = re.sub(pattern, replacement, content)

    # Replace Unicode escapes
    content = re.sub(r'\\u([0-9a-fA-F]{4})', decode_unicode_escape, content)

    return content

def calculate_line_column(source: str, position: int) -> Tuple[int, int]:
    """Calculate line and column for a position in source string."""
    if position >= len(source):
        position = len(source) - 1

    # Count lines from start to position
    line = 1
    column = 1

    for i, char in enumerate(source):
        if i >= position:
            break
        if char == '\n':
            line += 1
            column = 1
        elif char == '\r':
            # Handle \r\n as single newline
            if i + 1 < len(source) and source[i + 1] == '\n':
                continue
            line += 1
            column = 1
        else:
            column += 1

    return line, column
```

D. Core Logic Skeleton Code

`src/graphql_engine/ast/nodes.py` (Partial skeleton):

PYTHON

```python
"""AST node definitions using dataclasses."""

from dataclasses import dataclass, field
from typing import List, Optional, Any, Union
from .location import Location


@dataclass
class Node:
    """Base class for all AST nodes."""
    loc: Optional[Location] = None


@dataclass
class Document(Node):
    """Root node representing a complete GraphQL document."""
    definitions: List['Definition'] = field(default_factory=list)

    # TODO: Add method to get operation by name
    # TODO: Add method to get fragment by name


@dataclass
class Definition(Node):
    """Base class for definitions (operations, fragments)."""
    pass


@dataclass
class OperationDefinition(Definition):
    """Operation definition (query, mutation, subscription)."""
    operation_type: str = "query"  # "query", "mutation", "subscription"
    name: Optional[str] = None
    variable_definitions: List['VariableDefinition'] = field(default_factory=list)
    directives: List['Directive'] = field(default_factory=list)
    selection_set: 'SelectionSet' = field(default_factory=lambda: SelectionSet())

    # TODO: Add method to validate operation type is valid
    # TODO: Add method to get all variable names used


@dataclass
```

```python
class SelectionSet(Node):
    """Collection of fields/fragments within {...}."""
    selections: List['Selection'] = field(default_factory=list)

    # TODO: Add method to check if selection set is empty
    # TODO: Add method to get all field names in selection set

@dataclass
class Selection(Node):
    """Base class for selections (fields, fragment spreads, inline fragments)."""
    pass

@dataclass
class Field(Selection):
    """Field selection with optional alias, arguments, directives."""
    name: str = ""
    alias: Optional[str] = None
    arguments: List['Argument'] = field(default_factory=list)
    directives: List['Directive'] = field(default_factory=list)
    selection_set: Optional[SelectionSet] = None

    # TODO: Add property to get actual field name (alias or name)
    # TODO: Add method to get argument value by name

# TODO: Continue implementing all AST nodes from NAMING CONVENTIONS:
# - FragmentDefinition, FragmentSpread, InlineFragment
# - Argument, Directive, VariableDefinition, Variable
# - NamedType, ListType, NonNullType
# - All Value subclasses (IntValue, StringValue, ListValue, etc.)
# Use the exact field names from the naming conventions table

# Note: Use forward references as strings for types not defined yet
# Example: selection_set: Optional['SelectionSet'] = None
```

`src/graphql_engine/parser/lexer.py` (Partial skeleton):

PYTHON

```python
"""GraphQL lexer/tokenizer."""

from enum import Enum, auto
from dataclasses import dataclass
from typing import Optional
from ..error import GraphQLSyntaxError


class TokenType(Enum):
    """Types of GraphQL tokens."""
    EOF = auto()
    NAME = auto()
    INT = auto()
    FLOAT = auto()
    STRING = auto()
    BLOCK_STRING = auto()
    PUNCTUATION = auto()
    SPREAD = auto()
    # TODO: Add all token types from the Token Types Table


@dataclass
class Token:
    """A GraphQL token with value and location."""
    type: TokenType
    value: str
    line: int
    column: int

    def __repr__(self) -> str:
        return f"Token({self.type.name}, '{self.value}', {self.line}:{self.column})"


class Tokenizer:
    """Converts GraphQL source string into tokens."""

    def __init__(self, source: str):
        self.source = source
        self.position = 0
        self.line = 1
```

```
        self.column = 1
        self.current_char: Optional[str] = None
        self._advance()  # Initialize current_char

    def _advance(self) -> None:
        """Advance to next character in source."""
        # TODO: Implement character advancement
        # - Update position, line, column
        # - Handle \r\n as single newline
        # - Set current_char or None if EOF
        pass

    def _peek(self) -> Optional[str]:
        """Look at next character without consuming it."""
        # TODO: Return next character without advancing position
        pass

    def _skip_whitespace_and_comments(self) -> None:
        """Skip over whitespace and comments."""
        # TODO: Advance past spaces, tabs, commas, and #-comments
        # Comments run from # to end of line
        pass

    def _read_name(self) -> str:
        """Read a GraphQL name token."""
        # TODO: Starting from current_char (already validated as name start)
        # - Collect characters while is_name_continuation
        # - Return collected string
        pass

    def _read_number(self) -> Tuple[str, TokenType]:
        """Read integer or float literal."""
        # TODO: Handle:
        # - Optional leading minus sign
        # - Integer part
        # - Optional decimal point and fractional part
```

```
        # - Optional exponent (e/E with optional +/- and digits)
        # Return (value_string, TokenType.INT or TokenType.FLOAT)
        pass

    def _read_string(self) -> str:
        """Read regular string literal with escapes."""
        # TODO: Handle:
        # - Starting quote
        # - Characters with escape sequences
        # - Closing quote
        # - Use parse_string_value helper for escape processing
        pass

    def _read_block_string(self) -> str:
        """Read triple-quoted block string."""
        # TODO: Handle:
        # - Starting """
        # - Content with special whitespace handling
        # - Closing """
        # - Implement "trim indentation" algorithm from GraphQL spec
        pass

    def next_token(self) -> Token:
        """Return next token from source."""
        self._skip_whitespace_and_comments()

        if self.current_char is None:
            return Token(TokenType.EOF, "", self.line, self.column)

        start_line = self.line
        start_column = self.column

        # TODO: Implement token detection logic:
        # - Check for spread (...)
        # - Check for name start
        # - Check for number start (digit or - followed by digit)
```

```
        # - Check for string (")
        # - Check for block string (""")
        # - Otherwise single-character punctuation
        # Call appropriate _read_* method and return Token

        # TODO: For unknown character, raise GraphQLSyntaxError.from_lexer
        pass

    def tokenize(self) -> List[Token]:
        """Return all tokens from source."""
        tokens = []
        while True:
            token = self.next_token()
            tokens.append(token)
            if token.type == TokenType.EOF:
                break
        return tokens
```

`src/graphql_engine/parser/parser.py` (Partial skeleton):

PYTHON

```python
"""Recursive descent parser for GraphQL."""

from typing import List, Optional
from ..error import GraphQLSyntaxError
from ..ast.nodes import *
from .lexer import Tokenizer, Token, TokenType


class Parser:
    """Parses GraphQL documents from tokens."""

    def __init__(self, source: str):
        self.tokenizer = Tokenizer(source)
        self.tokens: List[Token] = []
        self.position = 0
        self._fill_tokens()

    def _fill_tokens(self) -> None:
        """Pre-load all tokens."""
        self.tokens = self.tokenizer.tokenize()

    def _peek(self) -> Token:
        """Look at current token without consuming it."""
        # TODO: Return token at current position or EOF if at end
        pass

    def _advance(self) -> Token:
        """Consume and return current token."""
        # TODO: Return current token and advance position
        pass

    def _expect(self, token_type: TokenType, value: Optional[str] = None) -> Token:
        """Expect and consume a specific token type."""
        # TODO: Check if current token matches expected type/value
        # If match: advance and return token
        # If no match: raise GraphQLSyntaxError.from_parser
        pass
```

```python
    def parse_document(self) -> Document:
        """Parse a complete GraphQL document."""
        # TODO: Implement according to parsing algorithm steps:
        # 1. Create empty definitions list
        # 2. While not at EOF: parse_definition and add to list
        # 3. Return Document(definitions)
        pass

    def parse_definition(self) -> Definition:
        """Parse a single definition (operation or fragment)."""
        # TODO: Check current token:
        # - If NAME and value is "fragment": return parse_fragment_definition
        # - If NAME and value is operation type: return parse_operation_definition
        # - If { (anonymous operation): return parse_operation_definition with default type
        # Otherwise: raise syntax error
        pass

    def parse_operation_definition(self, operation_type: str = "query") -> OperationDefinition:
        """Parse operation definition."""
        # TODO: Implement according to grammar:
        # 1. Parse optional name (if NAME token)
        # 2. Parse optional variable definitions (if '(')
        # 3. Parse optional directives
        # 4. Parse selection_set
        # 5. Return OperationDefinition
        pass

    def parse_selection_set(self) -> SelectionSet:
        """Parse {...} selection set."""
        # TODO: Implement recursive parsing:
        # 1. Expect '{'
        # 2. Create empty selections list
        # 3. While token ≠ '}': parse_selection and add to list
        # 4. Expect '}'
        # 5. Return SelectionSet(selections)
        pass
```

```
def parse_selection(self) -> Selection:
    """Parse field, fragment spread, or inline fragment."""
    # TODO: Check tokens:
    # - If ...: return parse_fragment (spread or inline)
    # - Otherwise: return parse_field
    pass


def parse_field(self) -> Field:
    """Parse field selection."""
    # TODO: Implement field parsing:
    # 1. Parse optional alias (NAME followed by ':')
    # 2. Parse field name
    # 3. Parse optional arguments
    # 4. Parse optional directives
    # 5. Parse optional selection_set
    # 6. Return Field with all components
    pass


# TODO: Implement remaining parsing methods:
# - parse_fragment_definition, parse_fragment_spread, parse_inline_fragment
# - parse_arguments, parse_directives, parse_variable_definitions
# - parse_value, parse_variable, parse_type
# - All value literal parsers (int, float, string, list, object, etc.)


def parse_query(self) -> Document:
    """Public API: Parse GraphQL query string."""
    try:
        return self.parse_document()
    except GraphQLSyntaxError:
        raise
    except Exception as e:
        # Wrap any unexpected error
        raise GraphQLSyntaxError(
            message=f"Unexpected parsing error: {str(e)}",
            locations=[]
```

```
            ) from e
```

`src/graphql_engine/parser/__init__.py` (Complete implementation):

```python
"""Public parser API."""

from .parser import Parser
from ..error import GraphQLSyntaxError


def parse_query(query_str: str) -> Document:
    """Parse GraphQL query string into AST Document.

    Args:
        query_str: GraphQL query/mutation/subscription string

    Returns:
        Document AST node

    Raises:
        GraphQLSyntaxError: If query has syntax errors
    """
    parser = Parser(query_str)
    return parser.parse_query()


def parse_schema(schema_str: str) -> List['TypeDefinition']:
    """Parse GraphQL Schema Definition Language.

    Args:
        schema_str: GraphQL SDL string

    Returns:
        List of type definitions

    Raises:
        GraphQLSyntaxError: If schema has syntax errors
    """
    # TODO: Implement in Milestone 2
    raise NotImplementedError("Schema parsing will be implemented in Milestone 2")
```

E. Language-Specific Hints

Python-Specific Implementation Tips:

1. **String Handling**: Use `str.isdigit()`, `str.isalpha()`, `str.isalnum()` for character classification. For Unicode names, Python's methods already handle Unicode correctly.
2. **Escape Sequences**: Use `re.sub()` with a dictionary of replacements for efficient escape sequence processing.
3. **Block Strings**: Implement the GraphQL spec's "trim indentation" algorithm:
   - Split into lines
   - Find common leading whitespace (excluding first/last lines)
   - Remove that whitespace from all lines
   - Join with `\n`
4. **Error Messages**: Include both the expected and actual token in error messages: `f"Expected '{expected}', got '{actual.value}'"`.
5. **Testing**: Use `pytest` with `@pytest.mark.parametrize` to test multiple query variations.
6. **Performance**: For educational code, clarity beats optimization. Use lists for token storage, simple loops for scanning.

F. Milestone Checkpoint

What to Test After Implementing Parser:

1. **Basic Queries**:

```bash
python -m pytest tests/test_parser.py::test_parse_simple_query -v
```

Expected: Test passes, AST contains correct operation type and field names.

2. **Error Detection**:

```bash
python -m pytest tests/test_parser.py::test_syntax_errors -v
```

Expected: Tests verify that malformed queries raise `GraphQLSyntaxError` with correct line/column.

3. **Fragment Handling**:

```bash
python -m pytest tests/test_parser.py::test_fragments -v
```

Expected: Named fragments and inline fragments parse correctly with type conditions.

4. **Manual Verification**:

```python
from graphql_engine.parser import parse_query


query = """
query GetUser($id: ID!) {
  user(id: $id) {
    name
    email @include(if: $showEmail)
  }
}
"""


ast = parse_query(query)
print(f"Operations: {len(ast.definitions)}")
print(f"Variables: {ast.definitions[0].variable_definitions}")
```

Expected Output: Shows 1 operation with 1 variable definition and 2 fields.

Signs Something Is Wrong:

- **Error messages point to wrong line/column**: Check location tracking in tokenizer
- **Fragments not being recognized**: Verify `...` token detection and `on` keyword handling
- **Nested fields cause infinite recursion**: Check `parse_selection_set` termination condition
- **String values have backslashes in them**: Escape sequence processing incomplete

Diagnostic Tools:

- Add `__repr__` methods to AST nodes for readable debugging
- Create a simple AST visualizer: `print_ast(ast, indent=0)`
- Log tokens during parsing: `print(f"Parsing {self._peek()}")`

Component 2: Schema & Type System

> **Milestone(s):** This section corresponds directly to Milestone 2: Schema & Type System, which focuses on building the foundational type system that defines what data can be queried and how it's structured. This component validates that the building plans (type definitions) are consistent and safe before any execution occurs.

Responsibility and Scope

The **Schema & Type System** component serves as the structural blueprint and validation engine for our GraphQL engine. Think of it as the city planning department that reviews architectural blueprints before construction begins—it ensures all buildings (types) follow zoning laws (GraphQL specification), connect properly via roads (relationships), and can safely accommodate residents (data).

Primary responsibilities include:

- **Type Definition Storage:** Representing all GraphQL type kinds (Object, Scalar, Enum, Interface, Union, InputObject, List, NonNull) in memory as structured data
- **Schema Construction:** Assembling individual type definitions into a coherent, validated `Schema` that serves as the single source of truth for what queries are valid

- **Type Validation:** Performing comprehensive consistency checks to detect issues like circular references, missing interface implementations, and type name conflicts
- **Introspection Implementation:** Providing the built-in `__schema`, `__type`, and `__typename` meta-fields that allow clients to query the schema structure itself
- **Type Resolution:** Mapping type names (like `"User"`) to their corresponding type definitions during query parsing and execution

Key boundaries and what this component does NOT do:

- Does NOT execute queries or call resolver functions (that's Component 3's responsibility)
- Does NOT generate types from databases (that's Component 4's responsibility)
- Does NOT validate query structure against schema (that's a separate validation phase)
- Does NOT handle runtime type coercions during execution (though it defines the rules for them)

The schema acts as a contract between the GraphQL service and its clients. Once built and validated, it becomes immutable for the duration of execution, ensuring consistent behavior.

Mental Model: Building Inspector

Imagine you're constructing a complex building with many rooms, hallways, and connections. The **type system** is like the complete set of architectural blueprints that define:

Blueprint Element	GraphQL Equivalent	Purpose
Room specifications	`ObjectType`	Defines what "furniture" (fields) exists in each room and how to access them
Room connections	`InterfaceType` / `UnionType`	Defines common hallways (interfaces) that connect similar rooms, or elevator banks (unions) that group different room types
Furniture inventory	`ScalarType`	Defines the basic building blocks (strings, numbers) that make up the furniture
Delivery instructions	`InputObjectType`	Defines how to deliver new furniture (input data) to the building
Building codes	Validation rules	Ensures blueprints don't create fire hazards (circular references) or illegal structures

The **schema building process** is the building inspector's review. They check that:

1. Every room has proper exits (interface implementations are complete)
2. No room leads to infinite hallways (no circular type references in input types)
3. All furniture fits through the doors (field argument types are compatible)
4. The building has a main entrance (`Query` type) and possibly service entrances (`Mutation` and `Subscription` types)

When introspection queries arrive, it's like giving visitors a self-guided tour map—the building inspector provides a simplified version of the blueprints that shows what's publicly accessible without revealing construction secrets.

Type System Interface

The type system exposes a clean API for defining types, building schemas, and querying type information. Here are the key interfaces:

Method/Function	Parameters	Returns	Description
`create_schema()`	`query_type: ObjectType`, `mutation_type: Optional[ObjectType]`, `subscription_type: Optional[ObjectType]`, `types: Optional[List[GraphQLType]]`, `directives: Optional[List[GraphQLDirective]]`	`Schema`	Constructs a validated schema from type definitions. Automatically includes introspection types and validates all type references.
`get_type(schema, name)`	`schema: Schema`, `name: str`	`Optional[GraphQLType]`	Looks up a type by name in the schema's type registry. Used internally during query validation and execution.
`is_input_type(type)`	`type: GraphQLType`	`bool`	Returns `True` if the type can be used as an input (scalar, enum, input object, or lists/non-nulls thereof). Critical for validating field arguments.
`is_output_type(type)`	`type: GraphQLType`	`bool`	Returns `True` if the type can be used as output (object, interface, union, scalar, enum, or lists/non-nulls thereof).
`assert_valid_schema(schema)`	`schema: Schema`	`None` (raises `GraphQLError` on failure)	Performs comprehensive validation of schema consistency. Called automatically by `create_schema()`.
`introspection_schema()`	(none)	`Schema`	Returns the built-in introspection schema that powers `__schema` and `__type` queries. This is merged with user-defined schemas.

The core data structures follow the naming conventions exactly. Here's the complete set of type definitions with all fields:

Type Name	Fields	Description
`GraphQLType`	`name: str`, `description: Optional[str]`	Base class for all GraphQL types. The `description` field supports documentation that appears in tools like GraphiQL.
`ScalarType`	Inherits from `GraphQLType` plus: `serialize: Callable`, `parse_value: Callable`, `parse_literal: Callable`	Represents leaf values like strings, numbers, or custom types. The three functions handle conversion between Python values, JSON values, and AST literals.
`ObjectType`	Inherits from `GraphQLType` plus: `fields: Dict[str, GraphQLField]`, `interfaces: List[InterfaceType]`	Represents a concrete object with fields. Each field has its own type and arguments. Objects can implement multiple interfaces.
`GraphQLField`	`type: GraphQLType`, `args: Dict[str, GraphQLArgument]`, `resolve: Optional[Callable]`, `description: Optional[str]`, `deprecation_reason: Optional[str]`	A field on an object type. The `resolve` function is stored here but only invoked by the execution engine.
`GraphQLArgument`	`type: GraphQLType`, `default_value: Any`, `description: Optional[str]`	An argument to a field. Default values are used when the argument isn't provided in the query.
`InterfaceType`	Inherits from `GraphQLType` plus: `fields: Dict[str, GraphQLField]`, `resolve_type: Optional[Callable]`	Abstract type defining a set of fields that implementing objects must include. `resolve_type` determines which concrete object type to use at runtime.
`UnionType`	Inherits from `GraphQLType` plus: `types: List[ObjectType]`, `resolve_type: Optional[Callable]`	Represents a type that could be one of several object types. Like interfaces but without shared fields definition.
`EnumType`	Inherits from `GraphQLType` plus: `values: Dict[str, EnumValue]`	Fixed set of allowed string values. The `EnumValue` includes `value: Any`, `description: Optional[str]`, and `deprecation_reason: Optional[str]`.
`InputObjectType`	Inherits from `GraphQLType` plus: `fields: Dict[str, GraphQLInputField]`	Used for complex inputs, like in mutations. Similar to objects but for input values only.
`GraphQLInputField`	`type: GraphQLType`, `default_value: Any`, `description: Optional[str]`	A field on an input object. Simpler than `GraphQLField` as it has no arguments or resolver.
`Schema`	`query_type: ObjectType`, `mutation_type: Optional[ObjectType]`, `subscription_type: Optional[ObjectType]`, `types: Dict[str, GraphQLType]`, `directives: Dict[str, GraphQLDirective]`	The complete schema containing all types. The `types` dictionary includes all types referenced in the schema (including built-in scalars and introspection types).

> **Key Insight:** The separation between `ObjectType` (for output) and `InputObjectType` (for input) is fundamental to GraphQL's design. Output types can have resolvers and support interfaces/unions, while input types must be serializable and cannot have those features. This distinction prevents confusing and potentially insecure patterns.

Schema Building Algorithm

Building a valid schema involves multiple validation passes to ensure internal consistency. Here's the step-by-step algorithm:

1. **Collect All Types**: Starting from the root operation types (`query_type`, `mutation_type`, `subscription_type`), recursively traverse all referenced types:
   - For each `ObjectType`, collect its fields' return types and argument types
   - For each `InterfaceType`, collect its fields' return types and argument types
   - For each `UnionType`, collect its member types

- For each field's arguments, collect the argument types
- For `ListType` and `NonNullType`, collect their wrapped types

2. **Validate Type Uniqueness**: Ensure no two types share the same name (case-sensitive). Built-in types (`Int`, `Float`, `String`, `Boolean`, `ID`, introspection types) are included and cannot be redefined.
3. **Validate Root Operation Types**:
   - The `query_type` must be an `ObjectType` and is required
   - If provided, `mutation_type` and `subscription_type` must be `ObjectType`
   - These types must exist in the collected types
4. **Validate Object Types** (for each `ObjectType`):
   - Validate each field's name is unique within the object
   - Validate field argument names are unique within each field
   - Validate field argument types are input types (`is_input_type` returns true)
   - Validate interfaces: for each interface the object claims to implement:
     - The interface must be an `InterfaceType`
     - The object must include all fields from the interface (name, type, and arguments must match exactly)
     - Interface field arguments must be identical (same name, type, and default value)
5. **Validate Interface Types** (for each `InterfaceType`):
   - Validate each field's name is unique within the interface
   - Validate field argument names are unique within each field
   - Validate field argument types are input types
   - Check for circular interface implementations (interface A cannot implement interface B if B implements A, directly or indirectly)
6. **Validate Union Types** (for each `UnionType`):
   - Must have at least one member type
   - All member types must be `ObjectType` (not interfaces, scalars, etc.)
   - Member type names must be unique within the union
7. **Validate Enum Types** (for each `EnumType`):
   - Must have at least one value
   - Enum value names must be unique within the enum
   - Values must follow GraphQL name rules (only letters, numbers, and underscore)
8. **Validate Input Object Types** (for each `InputObjectType`):
   - Validate each field's name is unique within the input object
   - Validate field types are input types
   - **Crucially, detect circular references**: No input object field can eventually reference itself through nested input objects. Use depth-first search with a visited set.
9. **Validate Scalar Types**: Built-in scalars are pre-defined. Custom scalars must provide all three conversion functions (`serialize`, `parse_value`, `parse_literal`).
10. **Validate Directives**: For each directive defined in the schema:
    - Validate directive name follows naming conventions
    - Validate directive locations are valid (QUERY, MUTATION, FIELD, etc.)
    - Validate directive arguments follow the same rules as field arguments
11. **Build Type Map**: Create a dictionary mapping type names to type instances, including all built-in types and introspection types. This map becomes the source of truth for type lookups.
12. **Mark Schema as Validated**: Once all validation passes, the schema is considered valid and immutable. Any attempt to modify types after validation should raise an error.

> **Design Principle:** Schema validation is idempotent. Calling `assert_valid_schema()` on an already-validated schema should return immediately. This allows the validation to be lazy but also eagerly checked during development.

Architecture Decision: Type Representation

> **Decision: Object-Oriented Type Hierarchy with Immutable Schemas**
>
> - **Context:** We need to represent all GraphQL type kinds in Python while maintaining type safety, enabling easy validation, and supporting introspection. The representation must be both human-friendly for schema builders and efficient for the execution engine.
> - **Options Considered:**
>   1. **Functional/Data-Centric Approach:** Types as plain dictionaries with `kind` fields, using functional validators
>   2. **Object-Oriented with Mutable Types:** Classes with inheritance, allowing types to be modified after creation
>   3. **Object-Oriented with Immutable Types:** Classes with inheritance where types are frozen after schema creation
> - **Decision:** Object-oriented hierarchy with immutable schemas (Option 3). Each type kind gets its own class inheriting from `GraphQLType`, and once a schema is built, its types cannot be modified.
> - **Rationale:**
>   - **Type Safety:** Python's class hierarchy provides natural "is-a" relationships that match GraphQL's type system (an `ObjectType` *is a* `GraphQLType`)
>   - **Performance:** Method dispatch via `isinstance()` checks is faster than dictionary `kind` checks
>   - **Immutability:** Prevents subtle bugs where types change during execution. Once validated, the schema is a reliable contract
>   - **Introspection:** Natural place to attach behavior (like validation methods) to type classes
> - **Consequences:**
>   - Slightly more boilerplate than dictionary approach
>   - Requires careful design of base classes to avoid deep inheritance
>   - Schema building becomes a two-phase process: create mutable type definitions, then freeze into immutable schema

Option	Pros	Cons	Why Not Chosen
Functional/Data-Centric	Simple serialization, minimal boilerplate	Type checking requires manual `kind` checks, harder to attach behavior	Lacks type safety and makes validation code more complex
OO with Mutable Types	Flexible for dynamic schema modifications	Risk of runtime errors if types change during execution	Mutability introduces race conditions and validation caching issues
OO with Immutable Types	**Type-safe, performant, reliable execution contract**	**Two-phase build process, slightly more code**	**Chosen for safety and performance**

Common Type System Pitfalls

Building a type system involves subtle traps that can lead to confusing errors or security issues. Here are the most common pitfalls and how to avoid them:

⚠️ Pitfall 1: Circular References in Input Types

- **Description:** Allowing input object fields that eventually reference themselves, like `type Input { self: Input }`. This creates infinite recursion during query validation and execution.
- **Why It's Wrong:** GraphQL explicitly forbids circular input references because they can't be serialized or validated. The specification states input types must be finite.
- **Fix:** During input object validation, perform a depth-first search from each input object, tracking visited types. If you encounter the same type again, raise a validation error with the circular path.

⚠️ Pitfall 2: Incomplete Interface Implementation

- **Description:** An object type claims to implement an interface but misses fields, has field type mismatches, or argument differences.
- **Why It's Wrong:** Breaks the Liskov Substitution Principle—clients expecting interface fields can't rely on them being present. Causes runtime errors when queries request interface fields on objects that don't implement them.

- **Fix:** When validating interface implementation, compare each interface field with the object's corresponding field:
  - Field names must match exactly
  - Field types must be identical (use deep equality, not just name comparison)
  - Arguments must match in name, type, and default value
  - Object can have additional fields not in the interface (that's allowed)

⚠️ Pitfall 3: Confusing Input and Output Types

- **Description:** Using output types (objects, interfaces, unions) as field arguments or input object fields, or using input types as field return types.
- **Why It's Wrong:** GraphQL's type system strictly separates input and output types for good reason. Input types must be serializable (no resolvers, no async), while output types can have behaviors.
- **Fix:** Implement `is_input_type()` and `is_output_type()` helpers that understand type wrappers (`ListType`, `NonNullType`). During validation:
  - Field argument types must be input types
  - Input object field types must be input types
  - Field return types must be output types

⚠️ Pitfall 4: Missing Nullable Wrapping Validation

- **Description:** Not validating that certain type positions (like list elements) can be nullable when they should be, or vice versa.
- **Why It's Wrong:** GraphQL has specific rules: lists can contain null values unless wrapped in NonNull, but fields can be made non-null. Getting this wrong causes unexpected null errors or missing data.
- **Fix:** Follow GraphQL's type modifier rules: `NonNullType` can wrap any type except another `NonNullType`. `ListType` can wrap any type. Validate these constraints when building types.

⚠️ Pitfall 5: Forgetting Built-in Introspection Types

- **Description:** Not including `__Schema`, `__Type`, `__Field`, `__InputValue`, `__EnumValue`, and `__Directive` in the type map, or not adding `__schema`, `__type`, and `__typename` fields.
- **Why It's Wrong:** Introspection is a required part of the GraphQL specification. Clients like GraphiQL rely on it to provide autocomplete and documentation.
- **Fix:** Always merge the built-in introspection schema with user-defined types. The `__typename` meta-field should be automatically added to every object and interface type.

Implementation Guidance

> **Technology Note:** For the type system component, we'll use Python's class system with `dataclasses` for clean representation and `typing` module for type hints. Since schemas are built once and read many times, we'll use immutable patterns after validation.

A. Recommended File Structure

```
graphql_engine/
├── __init__.py
├── type_system/          # Component 2: Schema & Type System
│   ├── __init__.py
│   ├── types.py          # All type class definitions
│   ├── schema.py         # Schema class and validation logic
│   ├── introspection.py  # Built-in introspection types and resolvers
│   ├── scalars.py        # Built-in scalar definitions
│   └── validation/       # Schema validation rules
│       ├── __init__.py
│       ├── rules.py      # Individual validation rules
│       └── validator.py  # Validation orchestration
└── parser/               # Component 1 (already built)
    └── ...
```

B. Infrastructure Starter Code

Here's complete, working code for the foundational type classes. Learners should place this in `type_system/types.py`:

```python
from __future__ import annotations

from typing import Any, Callable, Dict, List, Optional, Union

from dataclasses import dataclass, field

from enum import Enum


# Base type that all GraphQL types inherit from

@dataclass(frozen=True)

class GraphQLType:

    """Base class for all GraphQL types."""

    name: str

    description: Optional[str] = None


# Built-in scalar types

@dataclass(frozen=True)

class ScalarType(GraphQLType):

    """GraphQL scalar type definition."""

    serialize: Callable[[Any], Any] = field(repr=False)

    parse_value: Callable[[Any], Any] = field(repr=False)

    parse_literal: Callable[[Any], Any] = field(repr=False)


# Field definition for object and interface types

@dataclass(frozen=True)

class GraphQLField:

    """A field on a GraphQL object or interface type."""

    type: GraphQLType

    args: Dict[str, GraphQLArgument] = field(default_factory=dict)

    resolve: Optional[Callable[[Any, Any, Any, Any], Any]] = field(default=None, repr=False)

    description: Optional[str] = None

    deprecation_reason: Optional[str] = None


# Argument definition for fields and directives

@dataclass(frozen=True)

class GraphQLArgument:

    """An argument to a field or directive."""

    type: GraphQLType

    default_value: Optional[Any] = None

    description: Optional[str] = None


# Object type with fields and interfaces
```

```
@dataclass(frozen=True)
class ObjectType(GraphQLType):
    """GraphQL object type definition."""
    fields: Dict[str, GraphQLField] = field(default_factory=dict)
    interfaces: List[InterfaceType] = field(default_factory=list)

# Interface type definition
@dataclass(frozen=True)
class InterfaceType(GraphQLType):
    """GraphQL interface type definition."""
    fields: Dict[str, GraphQLField] = field(default_factory=dict)
    resolve_type: Optional[Callable[[Any, Any], str]] = field(default=None, repr=False)

# Union type definition
@dataclass(frozen=True)
class UnionType(GraphQLType):
    """GraphQL union type definition."""
    types: List[ObjectType] = field(default_factory=list)
    resolve_type: Optional[Callable[[Any, Any], str]] = field(default=None, repr=False)

# Enum value definition
@dataclass(frozen=True)
class EnumValue:
    """A value in a GraphQL enum type."""
    value: Any
    description: Optional[str] = None
    deprecation_reason: Optional[str] = None

# Enum type definition
@dataclass(frozen=True)
class EnumType(GraphQLType):
    """GraphQL enum type definition."""
    values: Dict[str, EnumValue] = field(default_factory=dict)

# Input field definition
@dataclass(frozen=True)
class GraphQLInputField:
    """A field on a GraphQL input object type."""
    type: GraphQLType
```

```
    default_value: Optional[Any] = None
    description: Optional[str] = None

# Input object type definition
@dataclass(frozen=True)
class InputObjectType(GraphQLType):
    """GraphQL input object type definition."""
    fields: Dict[str, GraphQLInputField] = field(default_factory=dict)

# Type modifiers (List and NonNull)
@dataclass(frozen=True)
class ListType(GraphQLType):
    """GraphQL list type wrapper."""
    of_type: GraphQLType = field(repr=True)
    # Override name to be derived from of_type
    def __post_init__(self):
        object.__setattr__(self, 'name', f'[{self.of_type.name}]')

@dataclass(frozen=True)
class NonNullType(GraphQLType):
    """GraphQL non-null type wrapper."""
    of_type: GraphQLType = field(repr=True)
    # Override name to be derived from of_type
    def __post_init__(self):
        object.__setattr__(self, 'name', f'{self.of_type.name}!')

# Schema definition
@dataclass(frozen=True)
class Schema:
    """Root GraphQL schema containing all types."""
    query_type: ObjectType
    mutation_type: Optional[ObjectType] = None
    subscription_type: Optional[ObjectType] = None
    types: Dict[str, GraphQLType] = field(default_factory=dict, repr=False)
    directives: Dict[str, GraphQLDirective] = field(default_factory=dict, repr=False)

# Directive definition
@dataclass(frozen=True)
class GraphQLDirective:
```

```
    """GraphQL directive definition."""

    name: str

    locations: List[str]

    args: Dict[str, GraphQLArgument] = field(default_factory=dict)

    description: Optional[str] = None
```

C. Core Logic Skeleton

Learners should implement the schema validation logic in `type_system/schema.py`:

PYTHON

```python
from typing import Dict, List, Optional, Set

from .types import *


def create_schema(
    query_type: ObjectType,
    mutation_type: Optional[ObjectType] = None,
    subscription_type: Optional[ObjectType] = None,
    additional_types: Optional[List[GraphQLType]] = None,
    directives: Optional[List[GraphQLDirective]] = None
) -> Schema:
    """
    Create and validate a complete GraphQL schema.

    Args:
        query_type: The root query type (required)
        mutation_type: Optional root mutation type
        subscription_type: Optional root subscription type
        additional_types: Extra types to include in the schema
        directives: Custom directives supported by the schema

    Returns:
        A validated, immutable Schema object

    Raises:
        GraphQLError: If the schema is invalid
    """
    # TODO 1: Collect all types starting from root operation types
    #   - Start with query_type, mutation_type, subscription_type
    #   - Recursively collect referenced types (field types, argument types, etc.)
    #   - Include built-in scalar types (Int, Float, String, Boolean, ID)
    #   - Include introspection types (from get_introspection_types())
    #   - Add any additional_types provided

    # TODO 2: Build type map: Dict[str, GraphQLType] mapping names to instances
    #   - Check for duplicate type names (case-sensitive)
    #   - Built-in types cannot be redefined
```

```
    # TODO 3: Validate all types according to GraphQL specification
    #   - Call validate_schema() with the complete type map
    #   - This should check: object types, interfaces, unions, enums, input objects

    # TODO 4: Create directives dictionary if directives provided

    # TODO 5: Return frozen Schema object with all validated components
    pass

def validate_schema(schema: Schema) -> None:
    """
    Validate a complete schema for consistency.

    This implements the validation rules from the GraphQL specification.
    Should be called automatically by create_schema().

    Raises:
        GraphQLError: With descriptive message about the first violation found
    """
    # TODO 1: Validate root operation types exist and are ObjectType

    # TODO 2: Validate all objects
    #   - Check field uniqueness
    #   - Check argument uniqueness and valid types
    #   - Verify interface implementations are complete

    # TODO 3: Validate all interfaces
    #   - Check field uniqueness
    #   - Check for circular interface implementations

    # TODO 4: Validate all unions
    #   - Must have at least one member type
    #   - Members must be ObjectType

    # TODO 5: Validate all enums
    #   - Must have at least one value
    #   - Value names must be unique
```

```python
    # TODO 6: Validate all input objects
    #   - Check field uniqueness
    #   - Verify no circular references (use depth-first search)

    # TODO 7: Validate directives
    #   - Check valid locations
    #   - Validate directive arguments

    pass

def is_input_type(type_: GraphQLType) -> bool:
    """
    Return True if the type can be used as an input type.

    Input types are: scalars, enums, input objects, and lists/non-nulls thereof.
    Output types (objects, interfaces, unions) cannot be used as inputs.
    """
    # TODO 1: Handle type modifiers (ListType, NonNullType)
    #   - Unwrap to the inner type and check recursively

    # TODO 2: Check the base type kind
    #   - Return True for: ScalarType, EnumType, InputObjectType
    #   - Return False for: ObjectType, InterfaceType, UnionType

    pass

def is_output_type(type_: GraphQLType) -> bool:
    """
    Return True if the type can be used as an output type.

    Output types are: objects, interfaces, unions, scalars, enums, and lists/non-nulls thereof.
    Input objects cannot be used as outputs.
    """
    # TODO 1: Handle type modifiers (ListType, NonNullType)

    # TODO 2: Check the base type kind
```

```
    #   - Return True for: ObjectType, InterfaceType, UnionType, ScalarType, EnumType
    #   - Return False for: InputObjectType


    pass
```

D. Language-Specific Hints

- **Use `@dataclass(frozen=True)`** for immutability. This prevents accidental modifications after schema creation.
- **Implement `__post_init__`** for computed fields like `ListType.name` that depends on `of_type.name`.
- **For circular reference detection**, use a depth-first search with a `visited` set. Python's recursion limit may be hit for very deep chains, so consider iterative approach or `sys.setrecursionlimit()`.
- **Use `typing.get_type_hints()`** for better introspection of your own types during development.
- **Consider `functools.lru_cache`** for expensive validation results that are called repeatedly.

E. Milestone Checkpoint

After implementing Component 2, learners should test their type system:

1. **Run the validation tests:**

```bash
python -m pytest graphql_engine/type_system/tests/ -v
```

2. **Expected output should show:**
    - All built-in scalar types are defined and have conversion functions
    - Simple schema creation succeeds with valid types
    - Invalid schemas raise descriptive `GraphQLError` messages
    - Introspection types are present in every schema
3. **Manual verification steps:**

```python
# Create a simple valid schema
from graphql_engine.type_system import create_schema, ObjectType, GraphQLField, ScalarType

# Define a simple object type
user_type = ObjectType(
    name="User",
    fields={
        "id": GraphQLField(type=ScalarType(name="ID")),
        "name": GraphQLField(type=ScalarType(name="String"))
    }
)

# Create schema
schema = create_schema(query_type=user_type)

# Verify introspection types exist
assert "__Schema" in schema.types
assert "__Type" in schema.types
assert "__typename" in user_type.fields  # Auto-added field

print("✓ Schema created successfully!")
```

4. **Signs something is wrong:**
   - **Error:** "Type 'String' not found" → Forgot to include built-in scalar types
   - **Error:** "Can't add field to frozen dataclass" → Trying to modify types after schema creation
   - **Error:** Maximum recursion depth exceeded → Circular reference detection not working
   - **Missing:** No `__typename` field on objects → Introspection field addition not implemented
5. **Debugging tips:**
   - Visualize the type hierarchy using the class diagram:

- Print the schema's type map to see all registered types
- Use `graphql_engine.parser.parse_schema()` to parse SDL and test against your type builder

Component 3: Query Execution Engine

> **Milestone(s):** This section corresponds to Milestone 3: Query Execution, which focuses on executing GraphQL queries against a schema, resolving fields, handling errors, and managing data fetching efficiently.

Responsibility and Scope

The **Query Execution Engine** is the core runtime that transforms parsed GraphQL queries into actual data. It's responsible for coordinating the entire resolution process from the root of the query down to every leaf field. The engine doesn't fetch data itself but orchestrates a network of **resolver functions** that do the actual work.

Responsibility	In Scope	Out of Scope
Resolver Invocation	Calling the appropriate resolver function for each field with correct arguments, parent value, and context.	Implementing business logic within resolvers (provided by user).
Error Propagation	Collecting field errors, maintaining partial results, and following GraphQL error spec for null propagation.	Retry logic or external error reporting (e.g., to monitoring services).
Parallel Execution	Executing independent sibling fields concurrently when possible to improve performance.	Distributed execution across multiple machines or microservices.
Null Handling	Enforcing GraphQL's nullability rules: non-null field errors bubble up to nearest nullable parent.	Database NULL value interpretation (handled by SQL compiler).
Argument Coercion	Converting raw variable/argument values (strings, numbers) to the expected GraphQL type (Int, Float, etc.).	Complex input validation beyond type correctness (e.g., business rules).
DataLoader Integration	Providing infrastructure for batching and caching to solve N+1 query problems.	Implementing specific batch loading logic for every data source.
Context Management	Passing a shared `ExecutionContext` object through all resolvers for request-scoped state.	Authentication/authorization logic (though context may hold user info).

The execution engine is purely synchronous in its core algorithm, but it can be extended to support asynchronous resolvers (returning promises/futures). Its primary output is an `ExecutionResult` containing the final data and any errors encountered.

Mental Model: Assembly Line

Think of query execution as an **automobile assembly line**. The raw materials (the query AST and root value) enter at the start. The assembly line has multiple stations (resolvers), each responsible for adding a specific part (field data) to the growing vehicle (result object). The conveyor belt moves the partially assembled vehicle from station to station.

Key parallels:

- **Work Order (Query AST):** The blueprint specifying exactly which parts to install and in what order.
- **Assembly Stations (Resolvers):** Specialized workers that know how to fetch or compute specific parts (field values).
- **Conveyor Belt (Execution Path):** The sequential flow from root fields down to nested selections.
- **Quality Control (Error Handling):** If a station fails to install a required part (non-null field error), the entire vehicle might be scrapped (null propagated upward), but optional parts can be missing without halting production.
- **Parallel Workstations (Sibling Fields):** Some stations can work simultaneously on different parts of the same vehicle (parallel field execution).

This mental model emphasizes that execution is a **coordinated, sequential process** where each step builds upon previous results, with strict rules about what happens when things go wrong.

Execution Interface

The public API of the execution engine is intentionally minimal. The primary entry point is a single function that accepts all necessary inputs and returns a complete result.

Method Signature	Returns	Description
`execute_query(schema, document_ast, variable_values=None, operation_name=None, context_value=None, root_value=None)`	`ExecutionResult`	Executes a GraphQL query/mutation against the provided schema. This is the main public interface.
`execute_query` Parameters		
`schema`	`Schema`	The GraphQL schema defining types and resolvers.
`document_ast`	`Document`	The parsed AST from `parse_query`.
`variable_values`	`Dict[str, Any]`	Values for variables defined in the operation.
`operation_name`	`Optional[str]`	Name of operation to execute if document contains multiple.
`context_value`	`Any`	Arbitrary value passed to all resolvers (e.g., database connection, user info).
`root_value`	`Any`	Initial parent value for root-level resolvers (often `None` or a special root object).

The `ExecutionContext` type is created internally and contains all the runtime state:

`ExecutionContext` Field	Type	Description
`schema`	`Schema`	Reference to the schema being executed against.
`document_ast`	`Document`	The parsed query document.
`operation_name`	`Optional[str]`	Name of the operation being executed.
`variable_values`	`Dict[str, Any]`	Values for variables after coercion.
`context_value`	`Any`	The context value passed from user (immutable during execution).
`root_value`	`Any`	The root value passed from user.
`fragments`	`Dict[str, FragmentDefinition]`	Map of fragment name to definition for quick lookup.
`errors`	`List[GraphQLError]`	Accumulated errors during execution.
`data_loaders`	`Dict[str, DataLoader]`	Map of DataLoader instances for batching.
`path`	`List[PathSegment]`	Current field path (list of field names/array indices) for error tracking.
`field_resolver`	`Callable`	Default function to resolve fields when no resolver is defined.
`type_resolver`	`Callable`	Default function to resolve concrete type for abstract types.

The `ExecutionResult` is the final output:

`ExecutionResult` Field	Type	Description
`data`	`Optional[Dict]`	The result data if execution succeeded (may be partial with errors).
`errors`	`List[GraphQLError]`	List of errors encountered during execution.
`extensions`	`Dict[str, Any]`	Optional extension metadata for profiling, tracing, etc.

Execution Algorithm

The execution follows a **depth-first, recursive resolution** pattern with special handling for null propagation and parallel execution. Below is the step-by-step algorithm for the main `execute_query` function:

1. **Operation Selection:**
    1. Extract all operation definitions from `document_ast.definitions`.
    2. If `operation_name` is provided, find the operation with that name. If not found, raise an error.
    3. If `operation_name` is `None` and there's exactly one operation, select it. If there are multiple operations, raise an error (must specify name).
    4. The selected operation must be either `query` or `mutation` (subscriptions are out of scope).
2. **Variable Coercion:**
    1. For each `VariableDefinition` in the operation's `variable_definitions`:
        - Get the provided value from `variable_values` dict.
        - If no value provided, check for `default_value`. If present, use it.
        - If no value and no default, and the variable type is non-null, raise an error.
        - **Coerce** the value to match the variable's GraphQL type using the type system's `parse_value` functions (for scalars) or recursive validation (for input objects/lists).
        - Store coerced value in the execution context's `variable_values`.
3. **Fragment Collection:**
    1. Gather all `FragmentDefinition` nodes from the document into a dictionary keyed by name for O(1) lookup during execution.
4. **Root Field Execution:**
    1. Determine the root `ObjectType` based on operation type: `schema.query_type` for queries, `schema.mutation_type` for mutations.
    2. Create initial `ExecutionContext` with all the prepared state.

3. Call `execute_operation(execution_context, operation, root_object_type, root_value)`.
    - **`execute_operation` algorithm:**
        1. Initialize an empty result map `{}`.
        2. Call `execute_selection_set(execution_context, operation.selection_set, root_object_type, root_value, result_map)`.
        3. Return `result_map`.

5. **Selection Set Execution (`execute_selection_set`):**
    1. For each `selection` in the selection set:
        - If selection is a `Field`:
            - Call `execute_field(execution_context, parent_type, parent_value, field)`.
        - If selection is a `FragmentSpread`:
            - Look up the fragment definition by name.
            - Check if the fragment's `type_condition` matches `parent_type` (using `does_fragment_condition_match`).
            - If it matches, recursively execute the fragment's `selection_set` via `execute_selection_set`.
        - If selection is an `InlineFragment`:
            - Check if the fragment's `type_condition` matches `parent_type` (or if no condition, it always matches).
            - If it matches, recursively execute the fragment's `selection_set`.

6. **Field Execution (`execute_field`):**
    1. **Field Definition Lookup:** Get the `GraphQLField` definition from `parent_type.fields[field.name]`.
    2. **Argument Coercion:** For each argument in the field definition's `args`, compute its value:
        - If the field AST has an `Argument` with matching name, use its value (coercing literals/variables).
        - Otherwise, use the argument's `default_value` if defined.
        - If no value and argument is non-null, raise an error.
    3. **Path Tracking:** Push the field's name (or alias) onto `execution_context.path`.
    4. **Resolver Selection:**
        - Use the field definition's `resolve` function if provided.
        - Otherwise, use the default `execution_context.field_resolver` (which typically returns a property from the parent object).
    5. **Resolver Invocation:** Call the resolver with arguments: `(parent_value, args, execution_context.context_value, info)` where `info` is an object containing field metadata, schema, and path.
    6. **Result Processing:**
        - If resolver raises an exception, catch it, add a `GraphQLError` to `execution_context.errors`, and set `result = None`.
        - If resolver returns a value, call `complete_value(execution_context, field.type, result)`.
    7. **Path Popping:** Pop from `execution_context.path`.
    8. Return the completed value.

7. **Value Completion (`complete_value`):**
    1. If `value` is `None`:
        - If the expected type is non-null (`NonNullType`), add an error and return `None` (error will bubble).
        - Otherwise, return `None`.
    2. If expected type is a `ListType`:
        - Ensure `value` is iterable. For each item, recursively call `complete_value` with the list's inner type.
        - Return a new list with completed items. If any item completion results in `None` (due to non-null error), the entire list becomes `None`.
    3. If expected type is a `ScalarType` or `EnumType`:
        - Call the type's `serialize` function to ensure the value is in the correct output format.
    4. If expected type is an `ObjectType`, `InterfaceType`, or `UnionType`:
        - Determine the concrete `ObjectType` (for interfaces/unions, use `resolve_type` callback).
        - Call `execute_selection_set` with this concrete type and the `value` as the new parent.
        - Return the resulting map.

8. **Parallel Execution:** Sibling fields (fields at the same nesting level in a selection set) have no implicit dependencies. The engine can execute them concurrently. In practice, this is implemented by:
    - Gathering all field promises/futures at a given level.
    - Waiting for all to complete before proceeding to nested selections.
    - This is especially powerful when combined with `DataLoader` batching.
9. **Result Assembly:** After the root operation completes, package the final `result_map` and collected `errors` into an `ExecutionResult` and return it.

> **Key Insight:** The algorithm's recursive structure mirrors the GraphQL query's hierarchical shape. Each level of recursion corresponds to a level of nesting in the query, with the `complete_value` function handling type-specific behavior at each step.

Architecture Decision: Execution Strategy

> **Decision: Synchronous Core with Async/Await Integration**
>
> - **Context:** GraphQL resolvers often need to fetch data from asynchronous sources (databases, APIs). The execution engine must support this without blocking. We must choose between a purely synchronous design, a callback-based async design, or leveraging modern async/await patterns.
> - **Options Considered:**
>   1. **Purely Synchronous:** All resolvers return plain values. Simple to implement but cannot handle I/O efficiently.
>   2. **Callback/Promise-based:** Resolvers return promises/tasks; engine manages a promise chain. Matches JavaScript's GraphQL.js but adds complexity.
>   3. **Async/Await Core:** Engine uses `async` / `await` throughout; resolvers can be sync or async. Leverages Python's `asyncio` natively.
> - **Decision:** Implement a **synchronous core** with clear extension points for async execution. The base algorithm assumes synchronous resolvers, but we provide an alternative `execute_query_async` entry point that uses `asyncio` for async resolvers. This keeps the learning curve gentle while allowing real-world use.
> - **Rationale:**
>   - **Educational Value:** Understanding the synchronous algorithm is foundational; async adds complexity that can be layered on later.
>   - **Language Support:** Python's `asyncio` is not trivial for beginners; offering both modes caters to different skill levels.
>   - **Performance:** For many use cases (especially with DataLoader batching), synchronous execution with threaded pools is sufficient.
> - **Consequences:**
>   - Learners must implement two execution paths if they want full async support.
>   - The synchronous engine cannot natively await async resolvers; they must be wrapped.
>   - The design allows incremental adoption: start with sync, then add async later.

Option	Pros	Cons	Chosen?
Purely Synchronous	Simple implementation, easy to debug, no event loop complexity.	Cannot efficiently handle I/O-bound resolvers, poor real-world performance.	No
Callback/Promise-based	Matches GraphQL.js pattern, enables non-blocking I/O.	Complex control flow, callback hell, difficult error handling.	No
Async/Await Core	Native Python support, clean syntax, excellent for I/O.	Requires `asyncio` understanding, all resolvers must be async, harder to debug.	No
Sync Core + Async Extension	Gentle learning curve, incremental adoption, supports both models.	Two execution paths to maintain, sync-to-async bridging required.	**Yes**

Common Execution Pitfalls

⚠️ Pitfall: Missing Null Propagation

- **Description:** When a non-null field resolver returns `None` or raises an error, the engine must "bubble up" the null to the nearest nullable parent field per the GraphQL spec. Forgetting this rule leads to incorrect partial results or type errors.

- **Why It's Wrong:** The spec is explicit: if a non-null field errors, its parent becomes `null` (if nullable), or continues bubbling. Violating this breaks client expectations and causes invalid response shapes.
- **Fix:** In `complete_value`, when encountering a `None` value for a `NonNullType`, add an error and return `None`. The caller (`execute_field`) must then propagate this `None` upward appropriately.

⚠️ Pitfall: Naive N+1 Query Problem

- **Description:** Resolving a list of objects, then resolving a related field on each object triggers separate database queries (N+1). This cripples performance.
- **Why It's Wrong:** Executing hundreds of queries for a single GraphQL request is inefficient and can overload the database.
- **Fix:** Implement the **DataLoader pattern**. Create a `DataLoader` per request that batches individual loader calls into a single batched query. The execution engine should provide a `data_loaders` dict in the context for resolvers to use.

⚠️ Pitfall: Improper Error Handling Swallowing Exceptions

- **Description:** If a resolver raises an exception and the engine doesn't catch it, the entire query fails with a generic 500 error, losing the specific error details.
- **Why It's Wrong:** GraphQL requires errors to be captured per-field and included in the `errors` array of the response, allowing partial data with errors.
- **Fix:** Wrap every resolver invocation in a try-catch. Convert caught exceptions to `GraphQLError` with appropriate `path` and `locations`. Add to `execution_context.errors` but continue executing other fields.

⚠️ Pitfall: Executing Resolvers in Wrong Order (Depth-First vs. Breadth-First)

- **Description:** The execution order matters for side effects and optimal batching. A naive depth-first approach might serialize independent sibling fields unnecessarily.
- **Why It's Wrong:** Missing opportunities for parallel execution reduces performance. For mutations, the spec requires sequential execution in the order they appear.
- **Fix:** Separate query and mutation execution. For queries, use a **breadth-first approach** for sibling fields to enable batching and parallelism. For mutations, enforce strict depth-first, left-to-right execution as per spec.

⚠️ Pitfall: Forgetting to Coerce Argument Values

- **Description:** Passing raw string/int values from variables directly to resolvers without converting them to the correct GraphQL type (e.g., string `"5"` to integer `5`).
- **Why It's Wrong:** Resolvers expect correctly typed arguments; mismatches cause runtime errors or incorrect behavior.
- **Fix:** In `execute_field`, for each defined argument, call `value_from_ast` (using the type system's `parse_literal`/`parse_value`) to convert AST values/variables to the appropriate Python type.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Concurrency	`threading.ThreadPoolExecutor` for parallel field execution.	`asyncio` with `async`/`await` for true async I/O.
DataLoader	Custom batching cache using dictionaries and a simple batch function.	Full `DataLoader` implementation with request-scoped caching, prime caching, and clear methods.
Error Tracking	Simple list of error dictionaries with message and path.	Full `GraphQLError` objects with extensions, original exception wrapping, and location formatting.
Resolver Info	Pass a simple dictionary with field name, args, etc.	Formal `ResolveInfo` class with schema, root value, operation, and field nodes.

B. Recommended File/Module Structure

```
graphql_engine/
│
├── execution/
│   ├── __init__.py
│   ├── executor.py           # Main synchronous Executor class
│   ├── executor_async.py     # Optional async Executor class
│   ├── values.py             # complete_value and type-specific completion
│   ├── errors.py             # GraphQLError class and error formatting
│   ├── dataloader.py         # DataLoader implementation
│   └── middleware.py         # Execution middleware (future extension)
│
├── schema/                   # From Component 2
├── parser/                   # From Component 1
└── utils/
    └── helpers.py
```

C. Infrastructure Starter Code

`errors.py` - Complete GraphQLError Implementation:

```python
from typing import Optional, List, Any, Dict

from dataclasses import dataclass


@dataclass

class Location:

    line: int

    column: int


@dataclass

class GraphQLError(Exception):

    message: str

    locations: Optional[List[Location]] = None

    path: Optional[List[str | int]] = None

    original_error: Optional[Exception] = None

    extensions: Optional[Dict[str, Any]] = None



    def __str__(self) -> str:

        return self.message



    def to_dict(self) -> Dict[str, Any]:

        """Convert error to GraphQL spec response format."""

        error_dict: Dict[str, Any] = {"message": self.message}

        if self.locations:

            error_dict["locations"] = [{"line": loc.line, "column": loc.column} for loc in self.locations]

        if self.path:

            error_dict["path"] = self.path

        if self.extensions:

            error_dict["extensions"] = self.extensions

        return error_dict
```

`dataloader.py` - Basic DataLoader Skeleton:

PYTHON

```python
from typing import Callable, List, Any, Dict, Optional
from collections import defaultdict

class DataLoader:
    def __init__(self, batch_load_fn: Callable[[List[Any]], List[Any]],
                 max_batch_size: Optional[int] = None):
        self.batch_load_fn = batch_load_fn
        self.max_batch_size = max_batch_size
        self.cache: Dict[Any, Any] = {}
        self.queue: List[Any] = []

    def load(self, key: Any) -> Any:
        """Schedule key for batch loading, return a future/promise."""
        # TODO 1: Check cache - if key exists, return cached value (could be promise)
        # TODO 2: Add key to queue for next batch
        # TODO 3: If queue length reaches max_batch_size, dispatch batch immediately
        # TODO 4: Return a promise that will resolve when the batch completes
        pass

    def dispatch(self) -> None:
        """Process all queued keys in a single batch."""
        # TODO 1: If queue is empty, return
        # TODO 2: Get unique keys from queue
        # TODO 3: Call batch_load_fn with list of keys
        # TODO 4: Match results to keys and resolve/reject each promise
        # TODO 5: Clear queue
        pass

    def clear(self, key: Any) -> None:
        """Remove key from cache."""
        if key in self.cache:
            del self.cache[key]

    def clear_all(self) -> None:
        """Clear entire cache."""
        self.cache.clear()
```

```
        self.queue.clear()
```

D. Core Logic Skeleton Code

`executor.py` - Main Executor Class:

PYTHON

```python
from typing import Dict, List, Any, Optional, Callable

from ..schema import Schema, ObjectType, GraphQLField, NonNullType, ListType

from ..parser import Document, OperationDefinition, Field, FragmentSpread, InlineFragment

from .errors import GraphQLError

from .dataloader import DataLoader


class ExecutionContext:

    # TODO: Define all fields from the Data Model table above

    # Hint: Use dataclass or simple class with __init__

    pass


class Executor:

    def execute_query(

        self,

        schema: Schema,

        document_ast: Document,

        variable_values: Optional[Dict[str, Any]] = None,

        operation_name: Optional[str] = None,

        context_value: Any = None,

        root_value: Any = None

    ) -> Dict[str, Any]:

        """

        Execute a GraphQL query synchronously.

        Returns a dict with 'data' and 'errors' keys per GraphQL spec.

        """

        # TODO 1: Validate schema is properly constructed (schema.types populated)

        # TODO 2: Select operation (query/mutation) based on operation_name

        # TODO 3: Coerce variable values to their declared types

        # TODO 4: Collect fragments into a dictionary by name

        # TODO 5: Create ExecutionContext with all parameters

        # TODO 6: Determine root operation type (query_type/mutation_type from schema)

        # TODO 7: Call execute_operation with root type and root_value

        # TODO 8: Package result and errors into ExecutionResult format

        # TODO 9: Return {'data': result, 'errors': formatted_errors}

        pass



    def execute_operation(
```

```
        self,
        context: ExecutionContext,
        operation: OperationDefinition,
        root_type: ObjectType,
        root_value: Any
    ) -> Any:
        """Execute a single operation (query or mutation)."""
        # TODO 1: Initialize empty result dict
        # TODO 2: Get selection set from operation
        # TODO 3: Call execute_selection_set with root_type and root_value
        # TODO 4: Return result dict
        pass

    def execute_selection_set(
        self,
        context: ExecutionContext,
        selection_set: List[Any],  # List of Selection nodes
        object_type: ObjectType,
        parent_value: Any,
        result_map: Dict[str, Any]
    ) -> None:
        """Execute all selections in a set, storing results in result_map."""
        # TODO 1: For each selection in selection_set:
        # TODO 2:   If selection is Field: call execute_field, store result at field's alias or name
        # TODO 3:   If selection is FragmentSpread: resolve fragment and execute its selection set
        # TODO 4:   If selection is InlineFragment: check type condition and execute its selection set
        # TODO 5: For query operations: execute sibling fields in parallel (use ThreadPoolExecutor)
        # TODO 6: For mutation operations: execute fields strictly sequentially
        pass

    def execute_field(
        self,
        context: ExecutionContext,
        parent_type: ObjectType,
        parent_value: Any,
        field_ast: Field
```

```
    ) -> Any:
        """Resolve a single field."""
        # TODO 1: Get field definition from parent_type.fields[field_ast.name]
        # TODO 2: Coerce field arguments using value_from_ast
        # TODO 3: Build resolve info object with field metadata, schema, etc.
        # TODO 4: Get resolver function (field.resolve or default_field_resolver)
        # TODO 5: Push field name onto context.path
        # TODO 6: Try: call resolver with (parent_value, args, context.context_value, resolve_info)
        # TODO 7: Catch Exception: convert to GraphQLError, add to context.errors, set result = None
        # TODO 8: Call complete_value with field.type and the result
        # TODO 9: Pop from context.path
        # TODO 10: Return completed value
        pass

    def complete_value(
        self,
        context: ExecutionContext,
        field_type: Any,  # GraphQLType
        result: Any
    ) -> Any:
        """Convert raw resolver result to proper GraphQL type output."""
        # TODO 1: If result is None: handle NonNullType error propagation
        # TODO 2: If field_type is ListType: iterate and complete each item
        # TODO 3: If field_type is ScalarType or EnumType: call serialize
        # TODO 4: If field_type is ObjectType: execute_selection_set on result
        # TODO 5: If field_type is InterfaceType or UnionType: resolve concrete type first
        # TODO 6: Return completed value
        pass
```

E. Language-Specific Hints

- **Default Field Resolver:** Implement a simple default that gets attributes from parent objects:

```
def default_field_resolver(parent_value, args, context_value, info):
    # If parent_value is a dict, return parent_value.get(info.field_name)
    # If parent_value is an object, return getattr(parent_value, info.field_name, None)
    # Otherwise return None
```

PYTHON

- **Parallel Execution:** Use `concurrent.futures.ThreadPoolExecutor` for synchronous parallel field execution. Remember that mutations must NOT run in parallel.

- **Value Coercion:** Leverage the `parse_value` methods on scalar types you built in Component 2. For input objects, recursively coerce each field.
- **Path Tracking:** Use a simple list for `context.path`. When entering a list item, append the index; when leaving, pop it.
- **Error Locations:** Extract `line` and `column` from the AST nodes' `loc` field to populate error locations.

F. Milestone Checkpoint

After implementing the execution engine, verify it works:

1. **Run Basic Tests:**

```bash
python -m pytest tests/execution/test_basic_resolution.py -v
```

Expected output: All tests pass, showing successful field resolution.

2. **Manual Test with a Simple Schema:** Create a test script:

```python
from graphql_engine import build_schema, execute_query

schema = build_schema("""
  type Query {
    hello: String
    user(id: ID!): User
  }
  type User {
    name: String
    age: Int
  }
""")

# Add resolver functions
schema.query_type.fields["hello"].resolve = lambda *_: "World"

result = execute_query(schema, "{ hello }")
print(result)  # Should show: {'data': {'hello': 'World'}, 'errors': []}
```

3. **Check for Common Issues:**
   - **Symptom:** Resolver exceptions crash the entire program.
     - **Diagnosis:** Missing try-catch in `execute_field`.
     - **Fix:** Wrap resolver call and convert exceptions to `GraphQLError`.
   - **Symptom:** Nested fields don't resolve (always return `null`).
     - **Diagnosis:** `complete_value` not calling `execute_selection_set` for object types.
     - **Fix:** Ensure object type completion triggers further execution.
   - **Symptom:** Non-null field errors don't bubble up.
     - **Diagnosis:** Missing null propagation logic in `complete_value`.
     - **Fix:** When a non-null field gets `None`, add error and return `None`.

4. **Verify Error Handling:**
    - Intentionally throw an exception in a resolver. The response should include `errors` array with details, but `data` may still have partial results.
    - Test non-null field: if it returns `None`, the parent field should become `null` (if nullable).
5. **Performance Check:**
    - Execute a query with 10 sibling fields. They should resolve concurrently (check timing).
    - For mutations, ensure fields execute sequentially even if they don't depend on each other.

Component 4: Database Schema Reflection

> **Milestone(s):** This section corresponds to Milestone 4: Database Schema Reflection, which focuses on auto-generating GraphQL schemas from database metadata by introspecting tables, columns, and relationships.

Responsibility and Scope

The **Database Schema Reflection** component acts as a bridge between the relational database world and the GraphQL type system. It transforms the flat, tabular structure of SQL databases into the hierarchical, strongly-typed GraphQL schema that Component 2 (Type System) can understand and Component 3 (Execution Engine) can query.

What this component DOES:

1. **Database Connection**: Establishes connections to target databases using appropriate drivers/adapters
2. **Metadata Introspection**: Queries database system catalogs (like `information_schema` or `pg_catalog`) to discover tables, columns, constraints, and relationships
3. **Type Mapping**: Converts SQL data types (INTEGER, VARCHAR, TIMESTAMP, etc.) to appropriate GraphQL scalar types (`Int`, `String`, `DateTime`, etc.)
4. **Relationship Detection**: Identifies foreign key constraints and translates them into GraphQL field relationships (one-to-one, one-to-many)
5. **Schema Generation**: Builds complete `ObjectType`, `InputObjectType`, and `EnumType` instances representing the database structure
6. **Query/Mutation Generation**: Creates root query fields for CRUD operations (find by ID, list with filtering/pagination) and mutation fields for create/update/delete operations
7. **Configuration Application**: Applies naming conventions, pluralization rules, and filtering based on configuration options

What this component does NOT do:

- Execute SQL queries (that's Component 5: Query to SQL Compilation)
- Validate GraphQL queries (that's the type system's responsibility)
- Handle authentication/authorization (explicit non-goal)
- Support real-time schema changes (uses static reflection at startup)

Key Data Structures This Component Owns:

Structure	Purpose	Relationship to Existing Types
`DatabaseMetadata`	Container for all introspected database information	New type specific to this component
`TableMetadata`	Information about a single database table	Used to generate `ObjectType`
`ColumnMetadata`	Information about a single column	Used to generate `GraphQLField`
`ForeignKeyMetadata`	Information about foreign key relationships	Used to generate relationship fields
`TypeMapping`	Mapping rules between SQL types and GraphQL types	Configuration-driven
`NamingConfig`	Rules for naming GraphQL types/fields from database names	Configuration-driven

Component Boundaries and Dependencies:

- **Input**: Database connection parameters (URL, credentials, schema name)
- **Output**: A complete `Schema` object ready for Component 3 to execute queries against
- **Depends On**: Component 2 (Type System) for creating type instances
- **Used By**: The main application to bootstrap the GraphQL server with auto-generated schema

Mental Model: Mirror

Think of database schema reflection as holding up a **mirror** to your database. The mirror doesn't create anything new—it simply reflects what's already there, but in a different format that GraphQL can understand. Just as a mirror shows your reflection but in reverse, schema reflection shows your database structure but transformed from tables/columns to types/fields.

Key Aspects of the Mirror Analogy:

1. **Faithful Representation**: A good mirror doesn't distort reality. Similarly, the reflection should accurately represent the database structure without hiding columns or changing data semantics.
2. **Perspective Shift**: When you look in a mirror, you see yourself from a different perspective. Similarly, reflection transforms the database from a relational perspective (tables with foreign keys) to a GraphQL perspective (types with nested fields).
3. **Selective Reflection**: Some mirrors are one-way or filtered. Our reflection can be configured to include/exclude certain tables, apply naming conventions, or hide sensitive columns.
4. **Static vs. Dynamic**: A traditional mirror shows a static reflection at a moment in time. Our reflection can be either static (captured once at startup) or dynamic (refreshed periodically), with important trade-offs.

Visualizing the Transformation:

```
Database Structure (Relational)      →      GraphQL Schema (Hierarchical)
─────────────────────────────────────       ──────────────────────────────

Table: users                         →       type User {
  id: INTEGER (PK)                           →    id: ID!
  name: VARCHAR(255)                         →    name: String
  email: VARCHAR(255)                        →    email: String
}                                            →    posts: [Post!]!   ← derived from FK

Table: posts                         →       type Post {
  id: INTEGER (PK)                           →    id: ID!
  user_id: INTEGER (FK → users.id)           →    title: String
  title: VARCHAR(255)                        →    content: String
  content: TEXT                              →    author: User!     ← derived from FK
}                                            → }
```

This mental model helps understand that we're not inventing a new schema—we're faithfully representing the existing database structure in GraphQL terms, with some opinionated transformations (like converting foreign keys to nested fields).

Reflection Interface

The reflection component exposes a clean, configuration-driven API that abstracts away database-specific details. Here's the complete public interface:

Primary Methods:

Method	Parameters	Returns	Description
`reflect_schema`	`connection_config: Dict[str, Any]` `options: ReflectionOptions`	`Schema`	Main entry point: connects to database, introspects metadata, builds and returns GraphQL schema
`get_database_metadata`	`connection_config: Dict[str, Any]` `filter: Optional[TableFilter]`	`DatabaseMetadata`	Low-level method that returns raw database metadata without building GraphQL types
`generate_types_from_metadata`	`metadata: DatabaseMetadata` `options: ReflectionOptions`	`Dict[str, GraphQLType]`	Converts database metadata to GraphQL types (can be used independently)
`build_schema_from_types`	`types: Dict[str, GraphQLType]` `options: ReflectionOptions`	`Schema`	Assembles types into a complete schema with query/mutation root types

Configuration Structure (`ReflectionOptions`):

Field	Type	Default	Description
`include_tables`	`List[str]`	`[]` (all)	Only reflect tables matching these patterns (supports SQL LIKE wildcards)
`exclude_tables`	`List[str]`	`[]`	Exclude tables matching these patterns (takes precedence over include)
`table_prefix`	`str`	`""`	Prefix to add to all table names when generating GraphQL type names
`table_suffix`	`str`	`""`	Suffix to add to all table names when generating GraphQL type names
`field_naming_convention`	`NamingConvention`	`CAMEL_CASE`	How to convert snake_case column names to GraphQL field names
`type_naming_convention`	`NamingConvention`	`PASCAL_CASE`	How to convert snake_case table names to GraphQL type names
`pluralization_rules`	`Dict[str, str]`	`{"y": "ies", ...}`	Rules for pluralizing type names in list fields
`type_mappings`	`Dict[str, ScalarType]`	Built-in defaults	Override default SQL-to-GraphQL type mappings
`include_relationships`	`bool`	`True`	Whether to generate fields for foreign key relationships
`max_relationship_depth`	`int`	`3`	Maximum depth for recursive relationship traversal (prevents cycles)
`include_comments`	`bool`	`True`	Use database column/table comments as GraphQL descriptions
`readonly`	`bool`	`False`	If true, only generate query fields (no mutations)

Connection Configuration Structure:

Field	Type	Required	Description
`database_type`	`str`	Yes	`"postgresql"`, `"mysql"`, `"sqlite"`, etc.
`host`	`str`	Depends	Database server hostname
`port`	`int`	Depends	Database server port
`database`	`str`	Yes	Database name
`schema`	`str`	No	Schema name (for databases with multiple schemas like PostgreSQL)
`username`	`str`	Depends	Authentication username
`password`	`str`	Depends	Authentication password
`connection_string`	`str`	Alternative	Raw connection string (alternative to individual parameters)
`ssl_mode`	`str`	No	`"disable"`, `"require"`, `"verify-ca"`, etc.
`connection_timeout`	`int`	No	Connection timeout in seconds

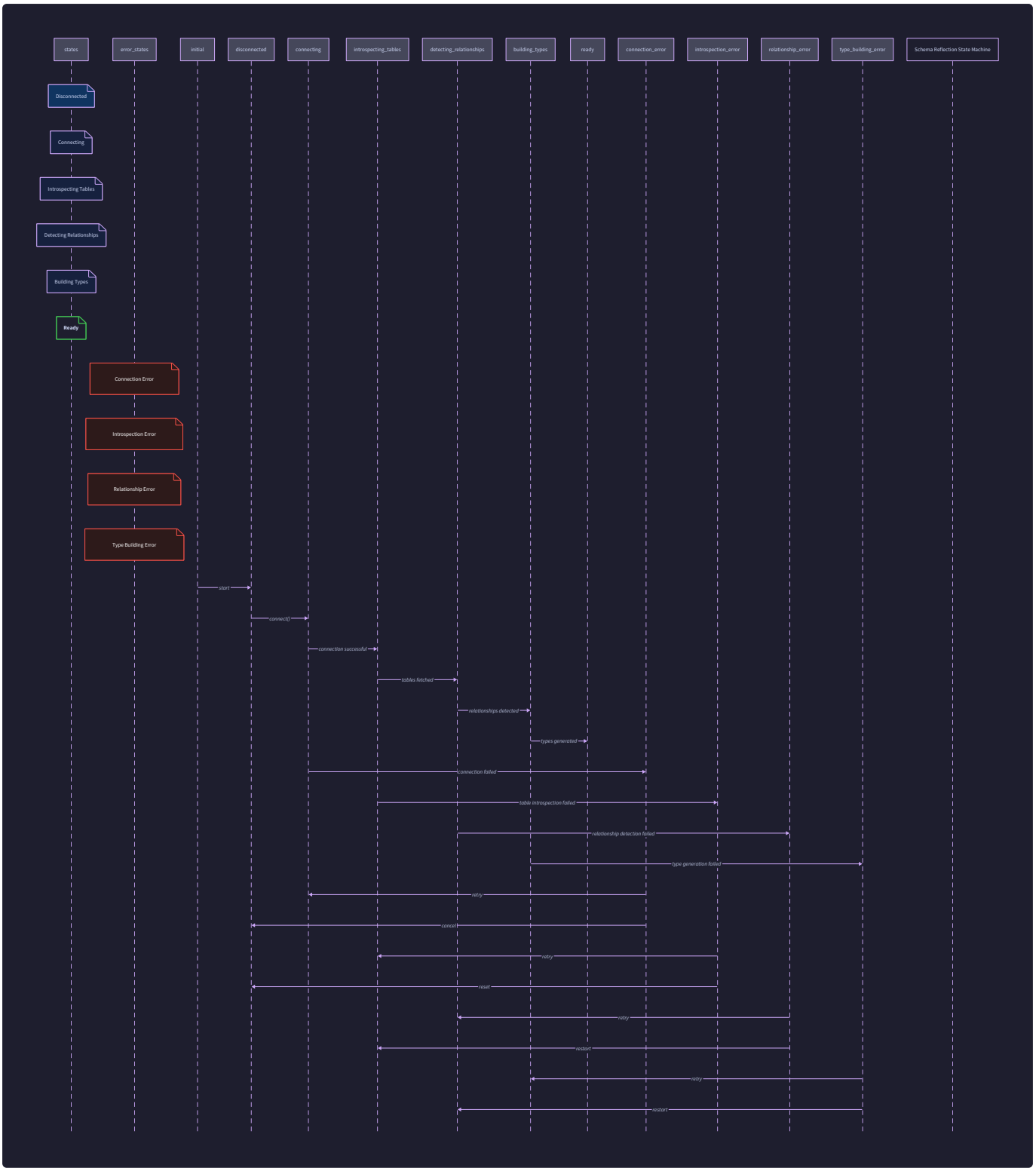
Usage Example in Prose: A developer wants to reflect a PostgreSQL database containing e-commerce tables. They would call:

```python
schema = reflect_schema(
    connection_config={
        "database_type": "postgresql",
        "host": "localhost",
        "port": 5432,
        "database": "ecommerce",
        "username": "app_user",
        "password": "secure_password"
    },
    options=ReflectionOptions(
        exclude_tables=["audit_%", "temp_%"],
        field_naming_convention=NamingConvention.CAMEL_CASE,
        type_naming_convention=NamingConvention.PASCAL_CASE,
        include_comments=True
    )
)
```

This would connect to the PostgreSQL database, introspect all tables except those starting with `audit_` or `temp_`, convert snake_case column names to camelCase field names, convert snake_case table names to PascalCase type names, and use database comments as GraphQL descriptions.

Reflection Algorithm

The reflection process follows a deterministic, multi-stage pipeline. Each stage transforms the data further toward a complete GraphQL schema. The state machine for this process is visualized in:



Algorithm Steps:

1. **Connection Establishment**

```
1.1. Validate connection configuration parameters
1.2. Load appropriate database adapter based on `database_type`
1.3. Establish connection using driver-specific method
1.4. Verify connection with a simple test query (e.g., `SELECT 1`)
1.5. If connection fails, raise `ConnectionError` with diagnostic information
```

2. **Metadata Collection**

```
2.1. Query `information_schema.tables` (or equivalent) for table list
2.2. Apply table filters (`include_tables`/`exclude_tables`)
2.3. For each table:
    2.3.1. Query `information_schema.columns` for column definitions
    2.3.2. Query `information_schema.key_column_usage` for primary/foreign keys
    2.3.3. Query `information_schema.table_constraints` for constraint types
    2.3.4. Query `pg_description`/equivalent for column/table comments (if `include_comments`)
2.4. Build `TableMetadata` objects with:
    - Table name, schema, comment
    - List of `ColumnMetadata` with name, type, nullable, default, comment
    - Primary key column names
    - List of `ForeignKeyMetadata` with target table/columns
```

3. **Type Mapping**

```
3.1. For each column in each table:
    3.1.1. Identify base SQL type (strip precision/length modifiers)
    3.1.2. Look up in type mapping table (default + custom overrides)
    3.1.3. Apply GraphQL type modifiers:
        - If column is nullable → plain type
        - If column is NOT NULL → wrap in `NonNullType`
        - If column is array type → wrap in `ListType`
    3.1.4. Store mapped GraphQL type in column metadata
3.2. Handle special cases:
    - Columns named `id` or ending in `_id` → map to GraphQL `ID` type
    - Timestamp/datetime columns → map to custom `DateTime` scalar
    - JSON/JSONB columns → map to custom `JSON` scalar
```

4. **Relationship Analysis**

```
4.1. For each foreign key constraint:
    4.1.1. Identify source table/columns and target table/columns
    4.1.2. Determine relationship cardinality:
        - If target columns include primary key → one-to-one or many-to-one
        - If source columns include primary key → one-to-one or one-to-many
    4.1.3. Check for self-referential relationships (table references itself)
    4.1.4. Detect join tables for many-to-many relationships
    4.1.5. Build relationship graph with cycle detection
    4.1.6. Stop traversing at `max_relationship_depth` to prevent infinite cycles
4.2. Build bidirectional field definitions:
    - For one-to-many: parent → children as `[ChildType!]!`
    - For many-to-one: child → parent as `ParentType!`
    - For one-to-one: bidirectional singular references
```

5. **GraphQL Type Generation**

```
5.1. For each table:
    5.1.1. Apply naming conventions to generate type name
    5.1.2. Create `ObjectType` with:
        - Name: transformed table name
        - Description: table comment (if available)
        - Fields: column fields + relationship fields
    5.1.3. Create `InputObjectType` for mutations:
        - Name: `Create[TypeName]Input` / `Update[TypeName]Input`
        - Fields: same as object type but with appropriate nullability
5.2. Generate enum types for:
    - Columns with CHECK constraints limiting to specific values
    - Columns with foreign keys to small reference/lookup tables
5.3. Register all types in type registry with proper dependency ordering
```

6. **Root Query/Mutation Generation**

```
6.1. Build root `Query` type with:
    6.1.1. For each table: `[typeName](id: ID!): TypeName` (single fetch)
    6.1.2. For each table: `all[pluralTypeName](filter: FilterInput, ...): [TypeName!]!` (list)
    6.1.3. Pagination arguments: `first`, `last`, `after`, `before`, `offset`, `limit`
    6.1.4. Filter arguments: per-column filter inputs with operators (eq, ne, lt, gt, etc.)
    6.1.5. Sort arguments: `orderBy: [TypeNameOrderBy!]`
6.2. If not `readonly`, build root `Mutation` type with:
    6.2.1. `create[TypeName](input: CreateInput!): TypeName!`
    6.2.2. `update[TypeName](id: ID!, input: UpdateInput!): TypeName!`
    6.2.3. `delete[TypeName](id: ID!): TypeName!`
6.3. Add aggregate fields: `count[TypeName](filter: FilterInput): Int!`
```

7. **Schema Assembly**

```
7.1. Create `Schema` object with:
    7.1.1. `query_type`: Generated Query type
    7.1.2. `mutation_type`: Generated Mutation type (if not readonly)
    7.1.3. `types`: All generated GraphQL types
    7.1.4. `directives`: Standard GraphQL directives
7.2. Run schema validation (using Component 2's `validate_schema`)
7.3. Return complete schema
```

Concrete Walk-Through Example:

Consider a simple blog database with `users` and `posts` tables. The algorithm would:

1. Connect to database and find tables: `users`, `posts`
2. Introspect columns:
   - `users`: `id` (INTEGER PK), `name` (VARCHAR), `email` (VARCHAR)
   - `posts`: `id` (INTEGER PK), `user_id` (INTEGER FK), `title` (VARCHAR), `content` (TEXT)
3. Detect foreign key: `posts.user_id` → `users.id` (many-to-one)
4. Map types: INTEGER → `Int`, VARCHAR → `String`, TEXT → `String`
5. Generate GraphQL types:
   - `User` with fields: `id: ID!`, `name: String`, `email: String`, `posts: [Post!]!`
   - `Post` with fields: `id: ID!`, `title: String`, `content: String`, `author: User!`
6. Generate root fields:
   - Query: `user(id: ID!): User`, `allUsers(...): [User!]!`, `post(id: ID!): Post`, `allPosts(...): [Post!]!`
   - Mutation: `createUser(...): User!`, `createPost(...): Post!`, etc.
7. Assemble and validate schema

Architecture Decision: Reflection Strategy

> **Decision: Static Reflection with Configuration-Driven Transformations**
>
> **Context**: We need to generate a GraphQL schema from a database schema, but databases and GraphQL have different structural paradigms (tables vs. types, foreign keys vs. nested fields). We must decide how aggressively to transform the database structure and whether to reflect dynamically (on each query) or statically (once at startup).
>
> **Options Considered**:
>
> 1. **Pure 1:1 Reflection**: Map each table to a type, each column to a field, foreign keys remain as scalar ID fields
> 2. **Aggressive Transformation**: Automatically create nested fields for relationships, generate pagination/filtering, apply naming conventions
> 3. **Dynamic Reflection**: Re-introspect database on each query or schema request
>
> **Decision**: We chose **Static Reflection with Configuration-Driven Transformations** (a balanced approach between options 1 and 2, with static rather than dynamic reflection).
>
> **Rationale**:
>
> - **Performance**: Static reflection happens once at startup, not on every query
> - **Predictability**: The schema is stable during server runtime
> - **Customizability**: Configuration options let users choose transformation aggressiveness
> - **Simplicity**: Static schemas are easier to debug and document
> - **Compatibility**: Works with GraphQL client tools that expect stable schemas
>
> **Consequences**:
>
> - ✅ Schema changes require server restart
> - ✅ Consistent performance regardless of database size
> - ✅ Can pre-compute optimized data structures
> - ❌ Doesn't automatically pick up database schema changes
> - ❌ Startup time increases with large databases
> - ❌ Memory usage for storing the complete reflected schema

Options Comparison Table:

Option	Pros	Cons	Why Not Chosen
Pure 1:1 Reflection	Simple implementation, no surprises	Poor GraphQL ergonomics (manual joins in queries), doesn't leverage GraphQL's strengths	Too simplistic—defeats purpose of GraphQL's nested queries
Aggressive Transformation	Best developer experience, automatic relationships	May create unexpected fields, harder to customize	Chosen as default but with configuration to dial back
Dynamic Reflection	Always up-to-date with database, no restarts needed	Performance overhead, caching complexity, unpredictable schema	Performance and stability concerns outweigh benefits
Static with Config (CHOSEN)	Balanced, configurable, performant, predictable	Requires restart for schema changes, memory overhead	Best trade-off for most use cases

Supporting Decisions:

Decision: Use Database System Catalogs Over ORM Models

Context: We need to introspect database structure. We could either query database system tables (`information_schema`) or use an ORM's model introspection.

Decision: Query database system catalogs directly.

Rationale:

- Works with any database without ORM model definitions
- Always reflects actual database state (not potentially stale model code)
- Consistent across different database backends (standardized `information_schema`)
- No dependency on specific ORM libraries

Consequences: Must handle database-specific quirks in system catalog queries.

Decision: Configuration-Driven Naming Conventions

Context: Database naming conventions (snake_case) differ from GraphQL conventions (camelCase). We must decide how to transform names.

Decision: Make naming conventions configurable with sensible defaults.

Rationale:

- Different teams have different preferences
- Legacy databases may have non-standard naming
- Can adapt to organizational standards
- Default (snake_case → camelCase) follows common GraphQL practice

Consequences: Configuration complexity increases but accommodates more use cases.

State Machine Transitions: The reflection process follows a defined state machine (referenced earlier). Here are the detailed transitions:

Current State	Event	Next State	Action Taken
`DISCONNECTED`	`connect_requested`	`CONNECTING`	Validate connection parameters, load database adapter
`CONNECTING`	`connection_succeeded`	`INTROSPECTING`	Establish connection, verify with test query
`CONNECTING`	`connection_failed`	`ERROR`	Log error details, clean up connection resources
`INTROSPECTING`	`metadata_collected`	`ANALYZING`	Query system catalogs, build metadata structures
`INTROSPECTING`	`introspection_failed`	`ERROR`	Handle permission errors, missing tables, etc.
`ANALYZING`	`analysis_complete`	`GENERATING`	Map types, detect relationships, build graph
`ANALYZING`	`circular_reference`	`ANALYZING`	Log warning, apply max depth limit, continue
`GENERATING`	`types_generated`	`ASSEMBLING`	Create GraphQL types from analyzed metadata
`ASSEMBLING`	`schema_built`	`READY`	Assemble complete schema, run validation
`ASSEMBLING`	`validation_failed`	`ERROR`	Log validation errors, suggest fixes
`READY`	`reset_requested`	`DISCONNECTED`	Close connections, clear cached metadata
Any state	`timeout`	`ERROR`	Operation took too long, abort and clean up

Common Reflection Pitfalls

⚠️ Pitfall: Incorrect Type Mapping

- **Description**: Mapping a SQL `DECIMAL(10,2)` to GraphQL `Float` loses precision, or mapping `TIMESTAMP WITH TIME ZONE` to `String` loses type safety.

- **Why It's Wrong**: Data loss, incorrect validation, poor client experience. GraphQL clients expect appropriate scalar types with proper coercion behavior.
- **How to Fix**: Create custom scalar types for database-specific types (e.g., `Decimal`, `DateTime`, `JSON`). Use database metadata about precision/scale to choose appropriate mappings.

⚠️ Pitfall: Missing Circular Relationship Detection

- **Description**: Database has self-referential foreign keys (e.g., `employee.manager_id → employee.id`) or circular chains (A → B → C → A), causing infinite recursion during type generation.
- **Why It's Wrong**: Infinite loops crash the reflection process or generate infinitely nested GraphQL types.
- **How to Fix**: Implement cycle detection in relationship graph traversal. Use `max_relationship_depth` configuration to limit traversal. For self-references, generate a nullable field to break the cycle.

⚠️ Pitfall: Naming Conflicts

- **Description**: Database has tables named `user` and `User` (case-insensitive in some databases but distinct in GraphQL), or a column name conflicts with a reserved GraphQL field name like `__typename`.
- **Why It's Wrong**: GraphQL schema validation fails with duplicate type/field names or reserved word conflicts.
- **How to Fix**: Apply name normalization (consistent casing), add prefixes/suffixes to conflicting names, rename reserved words (e.g., `type` → `type_`). Log warnings when conflicts are detected and automatically resolved.

⚠️ Pitfall: Permission Issues

- **Description**: Database user lacks permissions to query `information_schema` or specific tables, causing incomplete or empty reflection.
- **Why It's Wrong**: Generated schema is incomplete, missing tables or columns the application actually needs to access.
- **How to Fix**: Validate permissions during connection test. Provide clear error messages suggesting needed privileges. Offer `partial_reflection` mode that continues with accessible tables and logs warnings for inaccessible ones.

⚠️ Pitfall: Handling Database-Specific Quirks

- **Description**: PostgreSQL arrays, MySQL enum columns, SQLite type affinity, SQL Server schemas—each database has unique features not present in others.
- **Why It's Wrong**: Reflection fails or produces incorrect schemas when encountering unsupported features.
- **How to Fix**: Implement database adapters with specialized queries for each database type. Provide extension points for custom type handlers. Default to safe fallbacks (e.g., map unknown types to `String`).

⚠️ Pitfall: Ignoring Database Comments

- **Description**: Not using column/table comments from the database as GraphQL descriptions.
- **Why It's Wrong**: Misses opportunity for self-documenting API. GraphQL tools like GraphiQL show descriptions, improving developer experience.
- **How to Fix**: Always query for comments when available. Use empty string as default when comments are absent. Make this configurable (`include_comments` option).

⚠️ Pitfall: Generating Too Many Types

- **Description**: Reflecting every table including system tables, audit logs, temporary tables, or migration history tables.
- **Why It's Wrong**: Schema becomes bloated, confusing, and potentially exposes sensitive system information.
- **How to Fix**: Provide filtering options (`include_tables`, `exclude_tables`). Default to excluding common system table patterns. Allow regex or glob pattern matching.

Error Recovery Strategies:

Failure Mode	Detection	Recovery Action
Connection failure	Connection timeout or authentication error	Retry with exponential backoff (max 3 attempts), then raise clear error
Missing information_schema	Query returns empty or permission error	Fall back to database-specific system tables, log warning
Circular reference detected	Graph traversal depth exceeds limit	Stop traversing, make field nullable to break cycle, log warning
Type mapping not found	SQL type not in mapping table	Map to `String` scalar, log warning with suggestion for custom mapping
Name conflict	Duplicate normalized name detected	Add numeric suffix (`_2` , `_3`), log warning with both original names
Memory exhaustion	Memory usage exceeds threshold during large DB reflection	Process in batches, free intermediate data, log progress

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option	Rationale
Database Adapter	Single database support (PostgreSQL)	Pluggable adapters (PostgreSQL, MySQL, SQLite)	Start with one to learn pattern, extend later
Connection Pooling	No pooling (create per reflection)	Connection pooling with `aiopg` / `asyncpg`	Reflection happens rarely, pooling adds complexity
Metadata Caching	None (re-introspect each time)	File cache of serialized metadata	Speeds up development, reduces database load
Configuration	Python dictionary	Pydantic models with validation	Type-safe configuration with better error messages
Naming Utilities	Simple string replacement	inflection library (pluralize, camelize)	More robust handling of edge cases

B. Recommended File/Module Structure

```
graphql-engine/
├── reflection/                  # This component
│   ├── __init__.py
│   ├── core.py                  # Main reflection entry points
│   ├── metadata.py              # DatabaseMetadata, TableMetadata, etc.
│   ├── introspection/           # Database-specific adapters
│   │   ├── __init__.py
│   │   ├── base.py              # BaseIntrospector abstract class
│   │   ├── postgres.py          # PostgreSQL implementation
│   │   ├── mysql.py             # MySQL implementation
│   │   └── sqlite.py            # SQLite implementation
│   ├── mapping/                 # Type mapping logic
│   │   ├── __init__.py
│   │   ├── type_mapper.py       # SQL-to-GraphQL type mapping
│   │   ├── naming.py            # Naming convention utilities
│   │   └── defaults.py          # Default type mappings per database
│   ├── generators/              # GraphQL type generators
│   │   ├── __init__.py
│   │   ├── object_type.py       # Generate ObjectType from table
│   │   ├── input_type.py        # Generate InputObjectType
│   │   ├── query.py             # Generate root Query fields
│   │   └── mutation.py          # Generate root Mutation fields
│   ├── analysis/                # Relationship analysis
│   │   ├── __init__.py
│   │   ├── relationship.py      # Detect and analyze relationships
│   │   ├── graph.py             # Graph traversal for cycles
│   │   └── cardinality.py       # Determine 1:1, 1:many, many:many
│   └── config.py                # Configuration models
├── types/                       # Component 2 (Type System)
└── execution/                   # Component 3 (Execution Engine)
```

C. Infrastructure Starter Code

Complete Database Adapter Base Class:

PYTHON

```python
# reflection/introspection/base.py

from abc import ABC, abstractmethod

from typing import Dict, List, Optional, Any

from dataclasses import dataclass


@dataclass

class ColumnMetadata:

    """Metadata for a database column."""

    name: str

    table_name: str

    table_schema: str

    data_type: str

    is_nullable: bool

    is_primary_key: bool

    default_value: Optional[str]

    character_maximum_length: Optional[int]

    numeric_precision: Optional[int]

    numeric_scale: Optional[int]

    comment: Optional[str]


@dataclass

class TableMetadata:

    """Metadata for a database table."""

    name: str

    schema: str

    comment: Optional[str]

    columns: List[ColumnMetadata]

    primary_key_columns: List[str]

    foreign_keys: List[Dict[str, Any]]


@dataclass

class DatabaseMetadata:

    """Complete database metadata."""

    tables: Dict[str, TableMetadata]  # key: f"{schema}.{name}"

    enums: Dict[str, List[str]]       # enum name → list of values

    version: str                      # database version


class BaseIntrospector(ABC):

```

```
    """Abstract base class for database introspectors."""

    def __init__(self, connection_config: Dict[str, Any]):
        self.connection_config = connection_config
        self.connection = None

    @abstractmethod
    async def connect(self) -> None:
        """Establish database connection."""
        pass

    @abstractmethod
    async def introspect_tables(self,
                                schema: Optional[str] = None,
                                include_tables: Optional[List[str]] = None,
                                exclude_tables: Optional[List[str]] = None) -> DatabaseMetadata:
        """Introspect all tables in the database."""
        pass

    @abstractmethod
    async def get_table_comment(self, table_name: str, schema: str) -> Optional[str]:
        """Get comment/description for a table."""
        pass

    @abstractmethod
    async def get_column_comments(self, table_name: str, schema: str) -> Dict[str, str]:
        """Get comments for all columns in a table."""
        pass

    async def close(self) -> None:
        """Close database connection."""
        if self.connection:
            await self.connection.close()

    def __enter__(self):
        raise TypeError("Use async context manager")
```

```
    def __exit__(self, *args):
        pass

    async def __aenter__(self):
        await self.connect()
        return self

    async def __aexit__(self, *args):
        await self.close()
```

Complete Configuration Models with Validation:

PYTHON

```python
# reflection/config.py

from enum import Enum

from typing import Dict, List, Optional, Any

from pydantic import BaseModel, Field, validator


class NamingConvention(str, Enum):

    """Naming convention options."""

    CAMEL_CASE = "camel_case"       # column_name → columnName

    PASCAL_CASE = "pascal_case"     # column_name → ColumnName

    SNAKE_CASE = "snake_case"       # column_name → column_name (unchanged)

    KEBAB_CASE = "kebab_case"       # column_name → column-name


class ReflectionOptions(BaseModel):

    """Configuration options for schema reflection."""



    # Table filtering

    include_tables: List[str] = Field(default_factory=list)

    exclude_tables: List[str] = Field(default_factory=list)



    # Naming conventions

    table_prefix: str = ""

    table_suffix: str = ""

    field_naming_convention: NamingConvention = NamingConvention.CAMEL_CASE

    type_naming_convention: NamingConvention = NamingConvention.PASCAL_CASE



    # Type mapping

    type_mappings: Dict[str, Any] = Field(default_factory=dict)



    # Relationships

    include_relationships: bool = True

    max_relationship_depth: int = Field(default=3, ge=1, le=10)



    # Features

    include_comments: bool = True

    readonly: bool = False

    include_aggregates: bool = True

    include_filters: bool = True
```

```
    include_pagination: bool = True


    @validator('exclude_tables')
    def validate_filters(cls, v, values):
        """Ensure include/exclude don't conflict in confusing ways."""
        if values.get('include_tables') and v:
            # Warn if a table is both included and excluded
            conflicts = set(values['include_tables']) & set(v)
            if conflicts:
                import warnings
                warnings.warn(
                    f"Tables {conflicts} are both included and excluded. "
                    "Exclude takes precedence."
                )
        return v
```

D. Core Logic Skeleton Code

Main Reflection Entry Point:

PYTHON

```python
# reflection/core.py

from typing import Dict, Any, Optional

from .config import ReflectionOptions

from .metadata import DatabaseMetadata

from .introspection import get_introspector

from .mapping.type_mapper import TypeMapper

from .generators.object_type import generate_object_types

from .generators.query import generate_root_query

from .generators.mutation import generate_root_mutation

from types import Schema, ObjectType


async def reflect_schema(

    connection_config: Dict[str, Any],

    options: Optional[ReflectionOptions] = None

) -> Schema:

    """

    Main entry point: reflect database schema to GraphQL schema.



    Args:

        connection_config: Database connection parameters

        options: Configuration options for reflection



    Returns:

        Complete GraphQL Schema object



    Raises:

        ConnectionError: If database connection fails

        ReflectionError: If schema reflection fails



    Steps:

        1. Connect to database using appropriate introspector

        2. Collect metadata (tables, columns, constraints)

        3. Analyze relationships and build graph

        4. Map SQL types to GraphQL types

        5. Generate GraphQL types from metadata

        6. Generate root Query and Mutation types

        7. Assemble and validate complete schema
```

```
    """
    # TODO 1: Validate connection_config has required fields
    # TODO 2: Set default options if not provided
    # TODO 3: Get appropriate introspector for database type
    # TODO 4: Connect to database and collect metadata
    # TODO 5: Apply table filtering (include_tables/exclude_tables)
    # TODO 6: Detect and analyze relationships between tables
    # TODO 7: Map SQL types to GraphQL types using TypeMapper
    # TODO 8: Generate GraphQL ObjectType for each table
    # TODO 9: Generate root Query type with query fields
    # TODO 10: If not readonly, generate root Mutation type
    # TODO 11: Assemble Schema object with all types
    # TODO 12: Validate schema using Component 2's validate_schema
    # TODO 13: Return complete schema
    pass

async def get_database_metadata(
    connection_config: Dict[str, Any],
    schema: Optional[str] = None,
    include_tables: Optional[List[str]] = None,
    exclude_tables: Optional[List[str]] = None
) -> DatabaseMetadata:
    """
    Low-level method to get raw database metadata without building GraphQL types.

    Useful for debugging or custom schema generation pipelines.
    """
    # TODO 1: Get introspector for database type
    # TODO 2: Connect to database
    # TODO 3: Introspect tables with given filters
    # TODO 4: Return DatabaseMetadata object
    pass
```

Type Mapper with Default Mappings:

PYTHON

```python
# reflection/mapping/type_mapper.py

from typing import Dict, Any, Optional

from ..metadata import ColumnMetadata

from types import GraphQLType, ScalarType, ListType, NonNullType


class TypeMapper:

    """Maps SQL data types to GraphQL types."""



    # Default type mappings for PostgreSQL

    POSTGRES_MAPPINGS = {

        "integer": "Int",

        "bigint": "Int",

        "smallint": "Int",

        "numeric": "Decimal",  # Custom scalar

        "decimal": "Decimal",

        "real": "Float",

        "double precision": "Float",

        "text": "String",

        "varchar": "String",

        "char": "String",

        "boolean": "Boolean",

        "timestamp": "DateTime",

        "timestamptz": "DateTime",

        "date": "Date",

        "time": "Time",

        "json": "JSON",

        "jsonb": "JSON",

        "uuid": "ID",

        "bytea": "String",  # Base64 encoded

    }



    def __init__(self, custom_mappings: Optional[Dict[str, Any]] = None):

        self.mappings = self.POSTGRES_MAPPINGS.copy()

        if custom_mappings:

            self.mappings.update(custom_mappings)



    def map_column_type(self, column: ColumnMetadata) -> GraphQLType:
```

```
        """
        Map a database column to a GraphQL type.

        Args:
            column: Column metadata including data_type, is_nullable, etc.

        Returns:
            GraphQL type with appropriate modifiers (List, NonNull)

        Steps:
            1. Get base SQL type (strip length/precision modifiers)
            2. Look up in mappings table
            3. Apply NonNull wrapper if column is NOT NULL
            4. Apply List wrapper if column is array type
            5. Special handling for ID columns (name ends with _id or is primary key)
        """
        # TODO 1: Normalize SQL type (e.g., "varchar(255)" → "varchar")
        # TODO 2: Look up in self.mappings, fall back to "String" if not found
        # TODO 3: Get GraphQL scalar type from registry or create custom scalar
        # TODO 4: Check if column is array type (ends with [] in PostgreSQL)
        # TODO 5: If array, wrap in ListType
        # TODO 6: Check if column is nullable (is_nullable flag)
        # TODO 7: If NOT nullable, wrap in NonNullType
        # TODO 8: Special case: columns named 'id' or ending in '_id' → ID type
        # TODO 9: Return complete type with all modifiers
        pass

    def register_custom_mapping(self, sql_type: str, graphql_type: GraphQLType) -> None:
        """Register a custom SQL-to-GraphQL type mapping."""
        # TODO 1: Validate graphql_type is a scalar type
        # TODO 2: Add to mappings dictionary
        # TODO 3: Log warning if overwriting existing mapping
        pass
```

Relationship Detector:

PYTHON

```python
# reflection/analysis/relationship.py

from typing import Dict, List, Set, Tuple, Optional

from ..metadata import TableMetadata, DatabaseMetadata

from dataclasses import dataclass


@dataclass

class Relationship:

    """Represents a relationship between two tables."""

    source_table: str  # f"{schema}.{name}"

    source_columns: List[str]

    target_table: str

    target_columns: List[str]

    relationship_type: str  # "one_to_one", "one_to_many", "many_to_one", "many_to_many"

    foreign_key_name: Optional[str]

    is_self_referential: bool


class RelationshipAnalyzer:

    """Analyzes foreign keys to detect relationships between tables."""



    def __init__(self, max_depth: int = 3):

        self.max_depth = max_depth

        self.relationships: Dict[str, List[Relationship]] = {}

        self.reverse_relationships: Dict[str, List[Relationship]] = {}



    def analyze(self, metadata: DatabaseMetadata) -> Dict[str, List[Relationship]]:

        """

        Analyze all foreign keys to build relationship graph.



        Args:

            metadata: Database metadata with tables and foreign keys



        Returns:

            Dictionary mapping table name to list of relationships



        Steps:

            1. For each table, extract foreign key constraints

            2. Determine relationship cardinality
```

```
            3. Detect self-referential relationships
            4. Build adjacency lists for forward and reverse relationships
            5. Detect cycles and apply depth limiting
            6. Return complete relationship graph
        """
        # TODO 1: Initialize empty relationships dictionaries
        # TODO 2: For each table in metadata.tables
        # TODO 3: For each foreign key in table.foreign_keys
        # TODO 4: Determine source and target tables/columns
        # TODO 5: Check if self-referential (source == target)
        # TODO 6: Determine cardinality (check primary key involvement)
        # TODO 7: Create Relationship object
        # TODO 8: Add to relationships[source_table] and reverse_relationships[target_table]
        # TODO 9: After all FKs processed, detect many-to-many via join tables
        # TODO 10: Detect cycles using DFS with max_depth limit
        # TODO 11: Log warnings for circular references
        # TODO 12: Return relationships dictionary
        pass

    def _detect_cardinality(self,
                            source_table: TableMetadata,
                            target_table: TableMetadata,
                            source_columns: List[str],
                            target_columns: List[str]) -> str:
        """
        Determine relationship cardinality.

        Rules:
        - If target columns are primary key → many-to-one or one-to-one
        - If source columns are primary key → one-to-many or one-to-one
        - If both → one-to-one
        - If neither → many-to-many (requires join table detection)
        """
        # TODO 1: Check if target_columns ⊆ target_table.primary_key_columns
        # TODO 2: Check if source_columns ⊆ source_table.primary_key_columns
        # TODO 3: Return appropriate relationship_type string
```

```
    pass
```

E. Language-Specific Hints

Python-Specific Recommendations:

- **Async/Await**: Use `async` / `await` for database operations to avoid blocking the event loop. Most database drivers support async (e.g., `asyncpg` for PostgreSQL, `aiomysql` for MySQL).
- **Connection Management**: Use async context managers (`async with`) to ensure proper connection cleanup even on errors.
- **Type Hints**: Use Python's type hints extensively. This component has complex data structures that benefit from static type checking with mypy.
- **Pydantic**: Use Pydantic models for configuration validation—it provides excellent error messages and type coercion.
- **Caching**: Use `functools.lru_cache` for expensive operations like type mapping lookups.
- **Logging**: Use structured logging (`structlog` or `logging` with JSON formatter) to track reflection progress and issues.

Database-Specific Considerations:

- **PostgreSQL**: Use `asyncpg` for best performance. Query `pg_catalog` instead of `information_schema` for better performance with large schemas.
- **MySQL**: `information_schema` can be slow on large databases. Consider caching results.
- **SQLite**: No `information_schema` —use `PRAGMA table_info()` and `PRAGMA foreign_key_list()` .
- **SQL Injection**: Never concatenate table/column names into SQL queries. Use parameterized queries for values, but for identifiers, validate against a whitelist or use database-specific quoting functions.

Performance Optimizations:

- **Batch Queries**: Instead of querying each table's columns individually, batch queries using `UNION` or query all columns at once with appropriate filters.
- **Parallel Introspection**: Use `asyncio.gather()` to introspect multiple tables concurrently when safe to do so.
- **Metadata Caching**: Serialize `DatabaseMetadata` to JSON file after first reflection. On subsequent starts, check if database schema has changed (via `SELECT schema_version()` or similar) before using cache.

F. Milestone Checkpoint

After implementing Database Schema Reflection, you should be able to:

1. **Connect and Introspect**: Run a test that connects to your database and lists all tables:

```
python -m pytest tests/test_reflection.py::test_connect_and_list_tables -v
```
PYTHON

Expected output: Lists tables in your test database with column counts.

2. **Generate GraphQL Schema**: Run the reflection and output the generated GraphQL SDL:

```python
from reflection import reflect_schema
import asyncio

async def test_reflection():
    schema = await reflect_schema({
        "database_type": "postgresql",
        "database": "test_db"
    })
    print(schema.to_sdl())

asyncio.run(test_reflection())
```

Expected: Valid GraphQL Schema Definition Language showing types for your tables.

3. **Verify Relationships**: Check that foreign keys generate proper GraphQL fields:

```python
# After reflection, verify that User type has posts: [Post!]! field
# and Post type has author: User! field
```

4. **Test Configuration Options**: Apply different naming conventions and verify:

```python
# With camelCase: user_name → userName
# With PascalCase: user_name → UserName
```

Signs Something is Wrong:

- **No tables found**: Check database permissions, schema name, connection parameters
- **Missing relationships**: Check if foreign key constraints exist in database
- **Type mapping errors**: Unknown SQL types need custom mappings
- **Performance issues**: Large databases may timeout—increase timeout or implement paginated introspection

Verification Commands:

```bash
# Run unit tests for reflection component
pytest reflection/ -v

# Run integration test with actual database
pytest tests/integration/test_reflection_integration.py -v --db-url postgresql://user:pass@localhost/test_db

# Generate and validate schema
python scripts/generate_schema.py --validate

# Check for circular references
python scripts/analyze_relationships.py --max-depth 5
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Reflection returns empty schema	Database user lacks permissions	Check introspection queries manually: `SELECT * FROM information_schema.tables`	Grant SELECT on `information_schema` or specific tables
Missing foreign key relationships	Foreign keys not defined or introspector missing them	Query `information_schema.key_column_usage` directly	Ensure FKs exist in DB; check introspector SQL
GraphQL type name conflicts	Tables with same name in different schemas	Log normalized type names during generation	Use schema prefix: `SchemaName_TableName`
Memory usage spikes	Large database with many tables/columns	Monitor memory during reflection; use batch processing	Process tables in batches; free intermediate data
Reflection times out	Very large schema or slow database connection	Add timeout logging at each stage	Increase timeout; implement progress reporting
Custom type mapping not working	Mapping registered after introspection	Check order of operations in `reflect_schema`	Register custom mappings before calling `map_column_type`
Circular reference crashes	Self-referential FK without depth limit	Check logs for "max depth exceeded" warnings	Implement cycle detection; use `max_relationship_depth`
Generated schema invalid	Type validation fails	Run `validate_schema` on output; check error details	Fix type generation logic; ensure all types are registered

Diagnostic Tools:

1. **Metadata Inspector**: Write a simple script that dumps raw database metadata to JSON for inspection:

```python
metadata = await get_database_metadata(connection_config)

import json

print(json.dumps(metadata, default=str, indent=2))
```

2. **Relationship Visualizer**: Generate a DOT file of table relationships:

```python
analyzer = RelationshipAnalyzer()

relationships = analyzer.analyze(metadata)

# Output DOT format for Graphviz
```

3. **SQL Query Logger**: Enable query logging in your database adapter to see what introspection queries are being executed.

Common Configuration Issues:

- **Case sensitivity**: PostgreSQL is case-sensitive for quoted identifiers. If your tables were created with quotes (`CREATE TABLE "User"`), you must reference them as `"User"` not `user`.
- **Schema vs Database**: In PostgreSQL, `database.schema.table` vs MySQL's `database.table`. Ensure you're introspecting the correct schema.
- **Connection pools**: If using connection pooling, ensure connections have the right search path or default schema set.

Component 5: Query to SQL Compilation

> **Milestone(s):** This section corresponds to Milestone 5: Query to SQL Compilation, which focuses on translating GraphQL queries into optimized SQL statements that efficiently retrieve data from relational databases while avoiding common performance pitfalls.

Responsibility and Scope

The **Query to SQL Compilation** component bridges the conceptual gap between GraphQL's hierarchical, nested query model and relational databases' tabular structure. This component transforms a validated GraphQL query AST into one or more SQL statements that can be executed against a database, returning exactly the data requested by the client in the optimal format.

Core responsibilities:

- **AST-to-SQL translation:** Convert GraphQL field selections into SQL `SELECT` column lists
- **JOIN generation:** Translate nested object fields into appropriate SQL `JOIN` clauses based on foreign key relationships detected during schema reflection
- **Parameterization:** Convert GraphQL arguments and variables into SQL parameters to prevent injection vulnerabilities
- **Batching:** Combine multiple related data fetches into single SQL statements to avoid the N+1 query problem
- **Filtering and pagination:** Translate GraphQL filter arguments and pagination directives (`first`, `after`, `where`) into SQL `WHERE`, `LIMIT`, `OFFSET`, and `ORDER BY` clauses
- **Type coercion:** Ensure database column values are properly cast to GraphQL scalar types

Boundaries and limitations:

- Does **not** handle GraphQL subscriptions or real-time queries
- Does **not** implement database write operations (mutations) — only query compilation
- Does **not** handle database-specific optimizations like index hints or query plan analysis
- Relies on the **Database Schema Reflection** component for table/column metadata and relationship information
- Assumes the GraphQL schema has been validated and all types are properly defined

Data ownership: This component owns the transformation pipeline from GraphQL AST to parameterized SQL statements, including intermediate representations like `SQLSelect` and `SQLExpression` that capture the query structure before string generation.

Mental Model: Travel Planner

> Think of the SQL compiler as a **travel planner** for your data. When you want to visit multiple cities (tables) and see specific attractions (columns), a good travel planner doesn't book separate flights between each city (N+1 queries). Instead, they design an **efficient itinerary** (JOINs) that visits all desired locations in the optimal order, carrying only the luggage you need (selected columns), and following your specific preferences (filters and sorting).

Core analogy mappings:

Travel Planning Concept	SQL Compilation Concept
Destinations to visit	Tables referenced in the query
Attractions at each destination	Columns selected from each table
Travel routes between cities	Foreign key relationships enabling JOINs
Packing list (what to bring)	SELECT clause column list
Preferences (dates, budget)	WHERE clause filter conditions
Itinerary order	ORDER BY clause sorting
Trip duration limits	LIMIT/OFFSET pagination
Group tour vs. individual travel	Batched query vs. separate queries

The compiler's job is to analyze the complete "travel plan" (GraphQL query), identify all required destinations and their relationships, then generate the most efficient route that minimizes "travel time" (database roundtrips) while respecting all "traveler preferences" (query arguments).

Compiler Interface

The SQL compiler exposes a clean, focused API that accepts validated GraphQL queries and returns executable SQL statements with their parameters. The interface centers around the `compile_to_sql` method, which serves as the main entry point.

Public API Methods:

Method Name	Parameters	Returns	Description
`compile_to_sql`	`schema: Schema`, `document_ast: Document`, `operation_name: Optional[str] = None`, `variable_values: Dict[str, Any] = None`, `context: Dict[str, Any] = None`	`SQLQuery`	Main compilation entry point: converts a GraphQL query into an executable SQL statement with parameters
`analyze_selections`	`selection_set: SelectionSet`, `parent_type: GraphQLType`, `path: List[PathSegment]`, `table_alias: str`	`Tuple[List[SQLColumn], List[SQLJoin], List[RelationshipPath]]`	Internal helper: analyzes field selections and builds SQL column and join structures
`build_where_clause`	`arguments: List[Argument]`, `field_type: GraphQLType`, `table_metadata: TableMetadata`, `variable_values: Dict[str, Any]`	`Optional[SQLExpression]`	Constructs SQL WHERE conditions from GraphQL filter arguments
`apply_pagination`	`sql_select: SQLSelect`, `arguments: List[Argument]`, `variable_values: Dict[str, Any]`, `primary_key_columns: List[str]`	`SQLSelect`	Applies LIMIT, OFFSET, and ORDER BY clauses based on pagination arguments

The `SQLQuery` Data Structure: The compilation result encapsulates everything needed to execute the query against a database:

Field	Type	Description
`root_select`	`SQLSelect`	The main SELECT statement representing the root query operation
`nested_selects`	`List[SQLSelect]`	Additional SELECT statements for relationships that cannot be joined in a single query (e.g., many-to-many with additional filtering)
`parameters`	`List[SQLParameter]`	Positional or named parameters to be bound to the SQL statement
`result_mapper`	`Optional[Callable[[List[Dict]], Dict]]`	Function to transform flat SQL results into nested GraphQL response shape
`query_plan`	`Dict[str, Any]`	Execution plan metadata for debugging and optimization

Usage Pattern:

```python
# High-level usage example (conceptual, not code)

sql_query = compile_to_sql(

    schema=reflected_schema,

    document_ast=parsed_query,

    variable_values={"userId": 123},

    operation_name="GetUserWithPosts"

)


# Execute against database

results = db.execute(sql_query.root_select.sql, sql_query.parameters)

# Transform results using the mapper

nested_data = sql_query.result_mapper(results)
```

Compilation Algorithm

The compilation process follows a systematic, multi-phase approach that transforms GraphQL AST through several intermediate representations before producing executable SQL. The algorithm ensures correctness (returning exactly requested data) while optimizing for performance (minimizing database roundtrips).

Phase 1: Query Analysis and Normalization

1. **Operation Selection:** Identify the target operation from the `Document` AST based on `operation_name` or select the single operation if only one exists
2. **Variable Substitution:** Replace all GraphQL variables in the query with their actual values from `variable_values`, performing type coercion as needed
3. **Fragment Expansion:** Expand all fragment spreads (`...FragmentName`) and inline fragments (`... on Type`) into their full selection sets, filtered by type conditions
4. **Field Flattening:** Recursively traverse the selection set, building a flat list of field paths with their complete context (parent types, arguments, directives)

Phase 2: Join Planning and Relationship Analysis

5. **Root Table Identification:** Determine the primary database table corresponding to the GraphQL query root type (from schema reflection metadata)
6. **Relationship Graph Construction:** For each nested field in the selection set, identify the foreign key relationship between parent and child tables using the `Relationship` metadata from schema reflection
7. **Join Path Optimization:** Analyze all required joins and reorder them to:
   - Minimize the size of intermediate result sets (filter early)
   - Avoid cartesian products (ensure proper join conditions)
   - Respect database join limitations (some databases limit JOIN count)
8. **Join Type Selection:** Choose appropriate JOIN types:
   - `INNER JOIN` for non-nullable relationships
   - `LEFT JOIN` for nullable relationships or when the parent field might be null
   - `LATERAL JOIN` for one-to-many relationships with per-row subqueries (PostgreSQL)

Phase 3: SQL Generation

9. **Column Selection:** For each requested GraphQL field, map to the corresponding database column, applying any necessary type casts or transformations
10. **WHERE Clause Construction:** Convert GraphQL filter arguments into SQL conditions:
    - Equality checks (`{ where: { id: { eq: 123 } } }`) → `WHERE id = $1`

- Complex operators (`{ status: { in: ["ACTIVE", "PENDING"] } }`) → `WHERE status IN ($1, $2)`
- Nested object filters (`{ author: { name: { like: "%John%" } } }`) → `JOIN` with condition

11. **Sorting and Pagination:**
    - Convert `orderBy` arguments to `ORDER BY` clauses
    - Implement cursor-based pagination using `LIMIT` and `OFFSET` or `WHERE id > cursor` patterns
    - Apply `first` / `last` arguments as `LIMIT` with appropriate ordering
12. **Parameter Binding:** Replace all literal values with parameter placeholders (`$1` , `$2` , `?`) to prevent SQL injection, collecting values in the `parameters` list

Phase 4: Result Mapping Preparation

13. **Row-to-Object Mapping:** Generate a `result_mapper` function that knows how to transform the flat SQL result set (with columns from multiple joined tables) into the nested object structure expected by GraphQL
14. **Alias Handling:** Respect field aliases in the GraphQL query when constructing both the SQL column aliases and the result mapping
15. **Type Coercion Planning:** Plan any necessary type conversions from SQL result types to GraphQL scalar types (e.g., database `TIMESTAMP` → GraphQL `String` ISO format)

Concrete Walk-Through Example:

Consider a GraphQL query fetching a user and their posts:

```graphql
query GetUser($userId: ID!) {
  user(id: $userId) {
    id
    name
    email
    posts(first: 10, orderBy: CREATED_AT_DESC) {
      id
      title
      createdAt
    }
  }
}
```

The compilation algorithm processes this as:

1. **Operation Selection:** Identifies `GetUser` as the target operation
2. **Variable Substitution:** Replaces `$userId` with actual value (e.g., `123`)
3. **Fragment Expansion:** (No fragments in this query)
4. **Field Flattening:** Creates field paths: `user` (root), `user.id` , `user.name` , `user.email` , `user.posts` , `user.posts.id` , `user.posts.title` , `user.posts.createdAt`
5. **Root Table Identification:** Maps `User` GraphQL type to `users` database table
6. **Relationship Analysis:** Identifies `users.id` → `posts.user_id` foreign key relationship
7. **Join Planning:** Creates plan: `users` LEFT JOIN `posts` (since user might have no posts)
8. **Column Selection:** Maps fields to columns: `id` → `users.id` , `name` → `users.name` , etc.
9. **WHERE Clause:** Adds `WHERE users.id = $1`
10. **Pagination:** Adds `ORDER BY posts.created_at DESC LIMIT 10`
11. **Parameter Binding:** Binds `$1 = 123`
12. **Result Mapping:** Creates mapper that groups posts by user ID to nest them under each user

The final SQL might resemble:

```sql
SELECT users.id, users.name, users.email,
       posts.id AS posts__id, posts.title AS posts__title,
       posts.created_at AS posts__created_at
FROM users
LEFT JOIN posts ON users.id = posts.user_id
WHERE users.id = $1
ORDER BY posts.created_at DESC
LIMIT 10
```

Architecture Decision: Compilation Approach

Decision: Single-Query JOINs with Lateral Subqueries for Complex Cases

- **Context:** We must translate nested GraphQL queries to SQL while balancing performance (minimizing roundtrips) with correctness (handling complex filtering/pagination at each relationship level). Deeply nested queries with filtering at multiple levels present particular challenges.
- **Options Considered:**
  1. **Pure Single-Query JOINs:** Generate one massive SQL statement with all tables JOINed together
  2. **Separate Queries with Batching:** Issue separate SQL queries per relationship level but batch them using IN clauses
  3. **Hybrid with Lateral Joins:** Use single-query JOINs for simple cases, LATERAL JOINs for filtered one-to-many relationships
  4. **ORM-style N+1 then Optimize:** Start with naive N+1 queries, then optimize common patterns
- **Decision:** Option 3 (Hybrid with Lateral Joins) for PostgreSQL, with fallback to Option 2 (Batched Separate Queries) for databases without LATERAL JOIN support
- **Rationale:**
  - **Performance:** Single-query JOINs minimize database roundtrips (most important for network latency)
  - **Correctness:** LATERAL JOINs allow applying filters/limits to nested collections per parent row (unachievable with simple JOINs)
  - **Database Compatibility:** Fallback strategy ensures support across different SQL dialects
  - **Complexity Management:** Separating simple vs. complex cases keeps the common path simple
- **Consequences:**
  - **Positive:** Excellent performance for common query patterns, correct handling of nested filters/limits
  - **Negative:** More complex SQL generation logic, database-specific code paths, potential for very large SQL statements
  - **Mitigation:** Implement query size limits, provide EXPLAIN plan analysis for optimization hints

Comparison of Compilation Strategies:

Approach	Pros	Cons	Best For
Single-Query JOINs	- Single database roundtrip - Database optimizes entire query plan - Consistent ordering guarantees	- Cartesian product risk with many-to-many - Can't limit nested collections per parent - Large result sets with duplicates	Shallow queries without nested filters/limits
Batched Separate Queries	- Clean separation of concerns - Easy to apply per-level filters - Avoids duplicate data in results	- Multiple roundtrips (network overhead) - Connection pool pressure - Consistency timing issues	Deeply nested queries with complex per-level logic
LATERAL JOINs	- Single roundtrip - Correct per-parent filtering - Database optimizes across joins	- PostgreSQL-specific (not all DBs) - Complex SQL generation - Query planner may struggle	One-to-many relationships with filters/limits
GraphQL-to-SQL Middleware	- Reuse existing libraries - Community support - Battle-tested	- Less educational value - Hidden complexity - May not match our exact needs	Production systems (not learning projects)

Supporting Decision: Raw SQL Generation over ORM

- **Context:** We need to generate database queries that are both efficient and debuggable. Using an ORM provides abstraction but hides SQL details and limits optimization.
- **Decision:** Generate raw, parameterized SQL statements rather than using an ORM query builder
- **Rationale:**
  - **Transparency:** Learners can see exactly what SQL is generated and debug it
  - **Performance:** Direct control over SQL allows specific optimizations
  - **Flexibility:** Can use database-specific features and syntax
  - **Educational Value:** Teaches SQL generation patterns directly
- **Consequences:** Must handle SQL dialect differences, injection protection manually, and query string construction logic

Common Compilation Pitfalls

⚠️ Pitfall: Cartesian Products from Missing Join Conditions

- **Description:** Generating SQL with multiple JOINs but forgetting the ON conditions, causing every row from one table to match with every row from another (N×M results)
- **Why It's Wrong:** Explodes result set size exponentially, crashes database with memory exhaustion, returns incorrect data relationships
- **How to Fix:** Always verify every JOIN has an ON clause, use foreign key relationships from schema reflection, add validation that checks for Cartesian products in generated SQL

⚠️ Pitfall: N+1 Query Problem

- **Description:** For a list of N parent items, issuing N additional queries to fetch related child items instead of using a single JOIN or batched query
- **Why It's Wrong:** Horrible performance (N+1 roundtrips), database connection exhaustion, poor scalability
- **How to Fix:** Implement the DataLoader pattern for separate queries, or use JOINs/LATERAL joins for single-query approach. Always analyze the selection set to batch sibling field resolutions.

⚠️ Pitfall: SQL Injection Vulnerabilities

- **Description:** Concatenating user-provided values directly into SQL strings instead of using parameterized queries
- **Why It's Wrong:** Critical security vulnerability allowing attackers to execute arbitrary SQL, steal data, or destroy database
- **How to Fix:** Always use parameter binding (`$1` , `?`), never interpolate values. For dynamic column/table names (rarely needed), use an allowlist or strict validation.

⚠️ Pitfall: Missing Null Propagation Handling

- **Description:** When a LEFT JOIN returns NULL for child records, incorrectly treating the entire parent record as null rather than just the nested field

- **Why It's Wrong:** Violates GraphQL specification where nulls propagate to nearest nullable parent, returns incorrect empty results instead of partial data
- **How to Fix:** Implement proper null handling in the result mapper, distinguish between "parent not found" and "child relationship null". Use GraphQL's nullability rules from the schema.

⚠️ **Pitfall: Incorrect Type Mapping**

- **Description:** Database `DECIMAL(10,2)` values returned as strings instead of GraphQL Float, or database `TIMESTAMP` not converted to ISO string format
- **Why It's Wrong:** GraphQL clients receive data in wrong format, breaking type contracts and causing client-side errors
- **How to Fix:** Use the type mapping rules established during schema reflection consistently. Apply SQL CAST operations or post-processing transformations as needed.

⚠️ **Pitfall: Unbounded Result Sets**

- **Description:** Forgetting to add `LIMIT` clauses to queries that could return millions of rows, especially on list fields without `first` / `last` arguments
- **Why It's Wrong:** Database and application memory exhaustion, denial of service, poor performance
- **How to Fix:** Implement default limits (e.g., max 1000 rows), require pagination arguments for list fields, or implement query cost analysis to reject expensive queries.

⚠️ **Pitfall: Ignoring Database Indexes**

- **Description:** Generating WHERE clauses that don't align with database indexes, like applying functions to indexed columns (`WHERE LOWER(name) = 'john'`)
- **Why It's Wrong:** Forces full table scans instead of index seeks, terrible performance on large tables
- **How to Fix:** Analyze common filter patterns during schema reflection, generate WHERE clauses that preserve index usability, provide hints to create appropriate indexes.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
SQL Builder	String concatenation with careful escaping	SQLAlchemy Core or similar query builder library
Parameter Binding	`?` positional placeholders with list of values	Named parameters with dictionary binding
Database Adapter	DB-API 2.0 (Python's `sqlite3` , `psycopg2`)	Async database driver (`asyncpg` , `aiosqlite`)
Result Mapping	Manual dictionary transformation	Custom row-to-object mapper with metadata
Query Optimization	Basic join ordering heuristics	EXPLAIN plan analysis and cost-based optimization

B. Recommended File/Module Structure:

```
graphql_engine/
├── compiler/                          # SQL Compilation component
│   ├── __init__.py
│   ├── sql_builder.py                 # SQL string construction utilities
│   ├── join_planner.py                # JOIN analysis and optimization
│   ├── where_builder.py               # WHERE clause generation from filters
│   ├── pagination.py                  # Cursor-based pagination logic
│   ├── result_mapper.py               # Flat SQL → nested GraphQL transformation
│   ├── dialects/                      # Database-specific SQL generation
│   │   ├── __init__.py
│   │   ├── postgresql.py
│   │   ├── sqlite.py
│   │   └── mysql.py
│   └── tests/
│       ├── test_sql_builder.py
│       ├── test_join_planner.py
│       └── test_integration.py
├── execution/                         # Execution engine (from Component 3)
│   └── data_loader.py                 # DataLoader implementation for batching
└── reflection/                        # Schema reflection (from Component 4)
    └── metadata.py                    # Table/column metadata structures
```

C. Infrastructure Starter Code:

Complete SQL builder utilities for safe SQL generation:

PYTHON

```python
# graphql_engine/compiler/sql_builder.py

"""Safe SQL generation with parameter binding."""


from typing import List, Dict, Any, Optional, Union

from dataclasses import dataclass

from enum import Enum


class SQLDialect(Enum):

    POSTGRESQL = "postgresql"

    SQLITE = "sqlite"

    MYSQL = "mysql"


@dataclass

class SQLParameter:

    name: str

    value: Any

    type: str  # Optional type hint for the database


class SQLBuilder:

    """Builds parameterized SQL statements safely."""



    def __init__(self, dialect: SQLDialect = SQLDialect.POSTGRESQL):

        self.dialect = dialect

        self._parameters: List[SQLParameter] = []

        self._sql_parts: List[str] = []

        self._param_counter = 0



    def add_parameter(self, value: Any, type_hint: Optional[str] = None) -> str:

        """Add a value as a parameter and return its placeholder."""

        param_name = f"p{self._param_counter}"

        self._param_counter += 1

        self._parameters.append(SQLParameter(name=param_name, value=value, type=type_hint or "unknown"))



        if self.dialect == SQLDialect.POSTGRESQL:

            return f"${len(self._parameters)}"

        elif self.dialect == SQLDialect.SQLITE:

            return "?"

        elif self.dialect == SQLDialect.MYSQL:
```

```
            return "%s"
        else:
            return "?"


    def add_literal(self, sql_fragment: str) -> None:
        """Add a raw SQL fragment (for keywords, column names, etc.)."""
        self._sql_parts.append(sql_fragment)


    def add_identifier(self, identifier: str) -> None:
        """Add a properly quoted identifier (table/column name)."""
        if self.dialect == SQLDialect.POSTGRESQL:
            self._sql_parts.append(f'"{identifier}"')
        elif self.dialect == SQLDialect.MYSQL:
            self._sql_parts.append(f'`{identifier}`')
        else:
            self._sql_parts.append(f'"{identifier}"')


    def build(self) -> tuple[str, List[Any]]:
        """Return the SQL string and parameter values in correct order."""
        sql = " ".join(self._sql_parts)
        param_values = [p.value for p in self._parameters]
        return sql, param_values


    def clear(self) -> None:
        """Reset the builder for reuse."""
        self._parameters.clear()
        self._sql_parts.clear()
        self._param_counter = 0
```

D. Core Logic Skeleton Code:

Main compilation entry point with detailed TODOs:

PYTHON

```python
# graphql_engine/compiler/__init__.py

"""GraphQL to SQL compiler main entry points."""

from typing import Dict, Any, Optional, List

from ..ast import Document, OperationDefinition, Field, SelectionSet

from ..types import Schema, GraphQLType, ObjectType

from ..reflection.metadata import TableMetadata, Relationship

from .sql_builder import SQLBuilder, SQLDialect


def compile_to_sql(

    schema: Schema,

    document_ast: Document,

    operation_name: Optional[str] = None,

    variable_values: Optional[Dict[str, Any]] = None,

    context: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

    """

    Compile a GraphQL query to executable SQL.



    Args:

        schema: Validated GraphQL schema with database metadata

        document_ast: Parsed GraphQL query AST

        operation_name: Name of operation to execute (if multiple in document)

        variable_values: Values for GraphQL variables in the query

        context: Execution context (database connection, etc.)



    Returns:

        Dictionary with keys:

        - sql: The parameterized SQL string

        - parameters: List of parameter values to bind

        - result_mapper: Function to map SQL results to GraphQL shape

        - metadata: Compilation metadata for debugging

    """

    # TODO 1: Select the target operation from document_ast.definitions

    #   - If operation_name is None and there's exactly one operation, use it

    #   - If operation_name is specified, find the matching operation

    #   - Raise error if ambiguous or not found
```

```
# TODO 2: Extract and validate variable values
#   - Merge provided variable_values with operation's variable defaults
#   - Validate types match variable definitions
#   - Create a dict of resolved variable values for substitution

# TODO 3: Expand fragments in the selection set
#   - Gather all fragment definitions from the document
#   - Recursively replace fragment spreads with their actual selections
#   - Apply type conditions for inline fragments

# TODO 4: Analyze the root field and identify the target database table
#   - Get the GraphQL root type from schema (query_type)
#   - Look up the table metadata associated with this GraphQL type
#   - Determine the primary key columns for ordering/pagination

# TODO 5: Recursively analyze nested selections and build relationship paths
#   - For each field in the selection set, check if it's a relationship field
#   - Use schema reflection metadata to find foreign key relationships
#   - Build a tree of relationship paths from root to leaf fields

# TODO 6: Generate JOIN clauses for relationship paths
#   - Determine optimal join order (filter early, minimize intermediate size)
#   - Choose JOIN type (INNER vs LEFT) based on field nullability
#   - Generate ON conditions using foreign key relationships

# TODO 7: Build SELECT column list
#   - Map each GraphQL field to its database column
#   - Generate unique aliases for columns from joined tables
#   - Include any necessary type casts (timestamp → string, etc.)

# TODO 8: Build WHERE clause from arguments
#   - Extract filter arguments from the root field
#   - Convert GraphQL filter operators to SQL conditions
#   - Handle nested object filters (author: { name: { like: ... } })
#   - Use parameter binding for all values
```

```
    # TODO 9: Apply sorting and pagination
    #   - Parse orderBy arguments to ORDER BY clauses
    #   - Handle cursor-based pagination (after/before arguments)
    #   - Apply LIMIT based on first/last arguments with defaults

    # TODO 10: Generate the complete SQL statement
    #   - Use SQLBuilder to safely construct the SQL
    #   - Assemble: SELECT ... FROM ... JOIN ... WHERE ... ORDER BY ... LIMIT
    #   - Get final SQL string and parameter list

    # TODO 11: Create result mapper function
    #   - Analyze the selection tree structure
    #   - Generate a function that transforms flat SQL rows to nested objects
    #   - Handle field aliases and type coercion

    # TODO 12: Return compilation result
    #   - Include SQL, parameters, mapper function, and metadata
    #   - Add query plan information for debugging

    pass
```

Join planning algorithm skeleton:

PYTHON

```python
# graphql_engine/compiler/join_planner.py

"""Analyze relationships and plan optimal JOIN strategy."""


from typing import List, Dict, Any, Optional, Tuple

from dataclasses import dataclass

from ..reflection.metadata import Relationship, TableMetadata


@dataclass

class JoinPlan:

    """Represents a planned SQL JOIN."""

    left_table: str

    right_table: str

    join_type: str  # "INNER", "LEFT", "LATERAL"

    on_conditions: List[Tuple[str, str]]  # [(left_column, right_column), ...]

    relationship: Relationship

    alias: Optional[str] = None


def plan_joins(

    root_table: TableMetadata,

    relationships: Dict[str, List[Relationship]],

    selection_paths: List[List[str]]

) -> List[JoinPlan]:

    """

    Plan optimal JOIN order for accessing all tables in selection_paths.


    Args:

        root_table: The starting table (GraphQL query root)

        relationships: Dict mapping table names to their outgoing relationships

        selection_paths: List of field paths like ["user", "posts", "comments"]


    Returns:

        Ordered list of JoinPlan objects representing JOINs to execute

    """

    # TODO 1: Identify all tables needed based on selection_paths

    #   - Parse each path to determine which tables are accessed

    #   - Build set of required tables beyond the root


    # TODO 2: Find relationship paths from root to each required table
```

```
    #   - Use breadth-first search through relationship graph
    #   - Record the shortest path to each table
    #   - Handle multiple possible paths (choose based on cardinality)

    # TODO 3: Determine optimal join order
    #   - Start with root table
    #   - Add tables in order of increasing estimated row count (if metadata available)
    #   - Apply filters early: tables with WHERE conditions join earlier
    #   - Avoid cartesian products: ensure each JOIN has a condition

    # TODO 4: Determine JOIN type for each relationship
    #   - Use INNER JOIN for non-nullable relationships (GraphQL field is NonNull)
    #   - Use LEFT JOIN for nullable relationships
    #   - Consider LATERAL JOIN for one-to-many with per-row limits/filters

    # TODO 5: Generate ON conditions for each JOIN
    #   - Use foreign key column mappings from Relationship objects
    #   - For composite keys, multiple equality conditions
    #   - For self-referential relationships, use table aliases

    # TODO 6: Return ordered list of JoinPlan objects
    pass
```

E. Language-Specific Hints (Python):

- **Parameter Binding:** Use `%s` placeholders for MySQL, `?` for SQLite, `$1`, `$2` for PostgreSQL. Python's DB-API handles the differences.
- **Type Conversion:** For database-specific types, use SQL CAST: `CAST(column AS TEXT)` or Python-side conversion in the result mapper.
- **Connection Management:** Use connection pooling for production. For learning, simple connection-per-query is fine.
- **Async Support:** Consider `asyncpg` for PostgreSQL with async/await pattern if implementing async GraphQL execution.
- **SQL Injection Prevention:** Never use f-strings or `%` formatting with user input. Always use parameter binding.

F. Milestone Checkpoint:

What to Test:

1. **Basic Field Selection:** `{ users { id name } }` → `SELECT id, name FROM users`
2. **Nested Relationship:** `{ users { posts { title } } }` → SQL with JOIN between users and posts
3. **Filter Arguments:** `{ users(where: { status: { eq: "ACTIVE" } }) { id } }` → `WHERE status = $1`
4. **Pagination:** `{ users(first: 10, after: "cursor") { id } }` → `LIMIT 10 OFFSET ...`
5. **Parameter Binding:** Verify no raw values in SQL, all use placeholders

Expected Output:

```python
# Example test case

result = compile_to_sql(schema, parse_query("{ users { id name } }"))

print(result["sql"])

# Expected: SELECT id, name FROM users

print(result["parameters"])

# Expected: [] (empty list for simple query)


# With filter

result = compile_to_sql(schema, parse_query(

    '{ users(where: { id: { eq: 123 } }) { name } }'

))

print(result["sql"])

# Expected: SELECT name FROM users WHERE id = $1

print(result["parameters"])

# Expected: [123]
```

Verification Steps:

1. Run unit tests: `python -m pytest graphql_engine/compiler/tests/ -v`
2. Test with actual database: Execute generated SQL against test database, verify results match GraphQL expectation
3. Check for N+1: Use query logging to ensure nested fields don't cause additional queries
4. Security check: Attempt SQL injection through GraphQL variables, verify it's blocked

Signs Something is Wrong:

- ❌ SQL contains literal values instead of placeholders
- ❌ JOINs without ON clauses (cartesian product risk)
- ❌ Missing LIMIT on list fields (unbounded queries)
- ❌ Incorrect null handling (LEFT JOIN needed but using INNER)
- ❌ Type errors: database numbers returned as strings to GraphQL

Debugging Approach:

1. **Log generated SQL:** Add debug logging to see exactly what SQL is produced
2. **EXPLAIN plans:** Run `EXPLAIN ANALYZE` on generated SQL to check performance
3. **AST visualization:** Print the GraphQL AST to understand the selection structure
4. **Step-by-step compilation:** Test each compilation phase independently

---

Interactions and Data Flow

> **Milestone(s):** This section spans all five milestones, showing how the individual components connect to form a complete system. It illustrates the data transformations from GraphQL string to database query and back, and how schema reflection integrates with the type system.

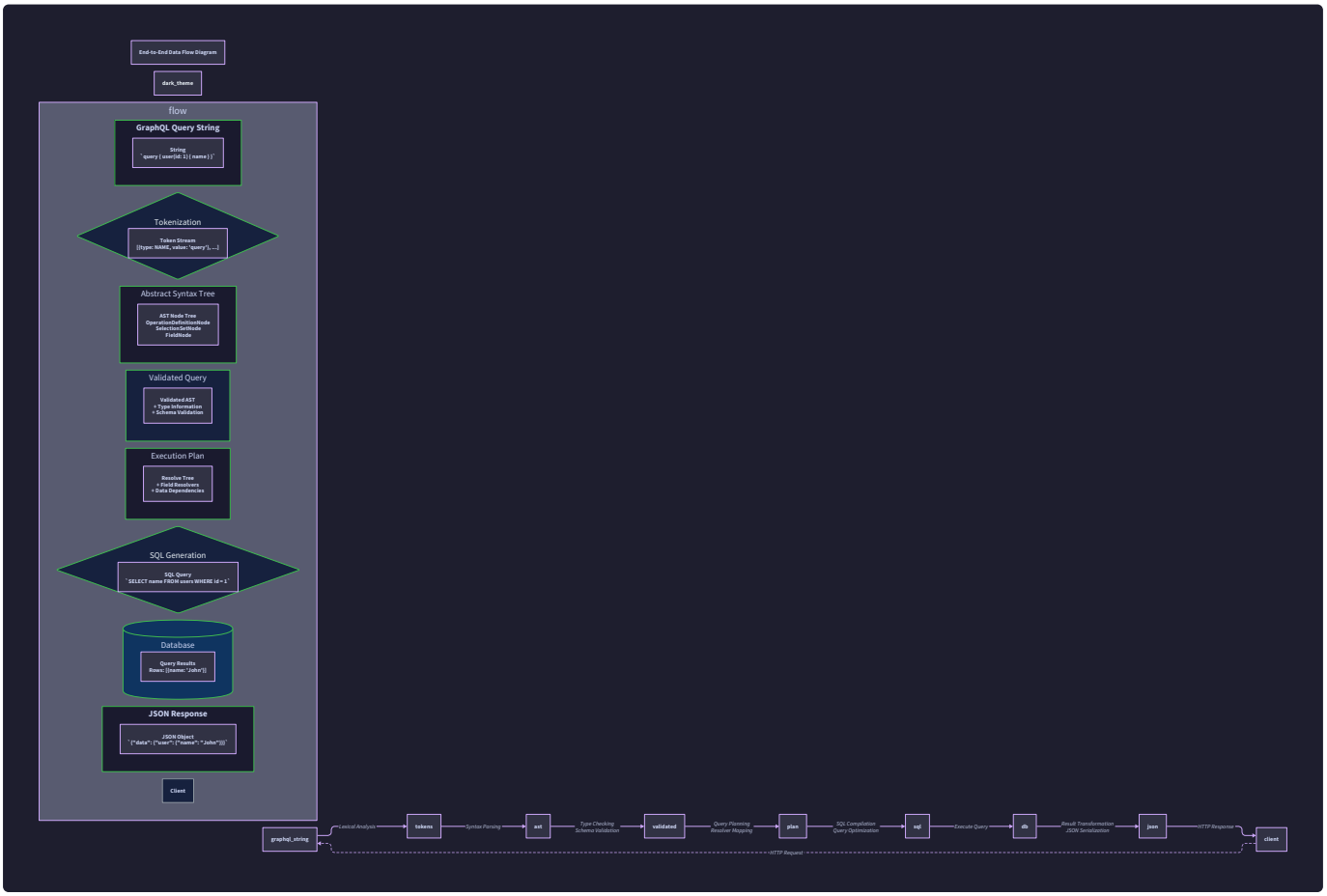
Understanding how components interact is crucial for seeing the system as more than isolated parts. This section maps the journey of a GraphQL query through the engine, reveals how the database schema becomes a GraphQL type system, and details the precise data structures exchanged between components. Think of it as following a package through a logistics network: we trace its path, document its transformations at each hub, and understand the routing decisions that determine its final delivery.

Happy Path: Query Execution Flow

The happy path represents a successful GraphQL query execution from HTTP request to JSON response. This flow integrates components from Milestones 1, 2, 3, and 5, demonstrating how parsed queries become execution plans and finally database results.

> **Mental Model: Assembly Line with Quality Control**
> Imagine an automotive assembly line. A raw query (sheet metal) enters the factory. The parser (stamping press) shapes it into a structured AST (car frame). The type system (quality inspection) validates the frame against blueprints. The SQL compiler (robot arm) welds on the engine and wheels (SQL generation). The execution engine (final assembly) installs the interior (resolves fields) and the database (engine) provides power. At each station, defects are caught and flagged, but the line continues for other cars (fields).

The end-to-end flow follows this sequence, which you can visualize in



:

Step 1: HTTP Request Ingestion The process begins when an HTTP server (not part of our core engine) receives a POST request containing a GraphQL query string, optional variables, and an operation name. The server extracts these components and passes them to the engine's main entry point.

Step 2: Query Parsing (Milestone 1) The `parse_query` function is invoked with the query string. Internally, this triggers:

1. **Lexical Analysis**: The `Tokenizer.tokenize()` method scans the string, producing a list of `Token` objects (names, punctuation, literals) with `Location` data.
2. **Syntax Parsing**: The `Parser.parse_document()` method consumes the token stream using recursive descent, building a complete `Document` AST. This `Document` contains one or more `Definition` nodes (typically an `OperationDefinition`).

Transformation	Input Format	Output Format	Key Operation
Raw string	`"query { user(id:1) { name } }"`	`Document` with `OperationDefinition`	Tokenization and recursive descent parsing

Step 3: Schema Preparation The engine obtains a `Schema` object. This can come from:

- **Static definition**: Built programmatically using `create_schema()` with hand-defined types.
- **Dynamic reflection**: Generated via `reflect_schema()` from a database (integration with Milestone 4). The schema is already validated and ready for execution.

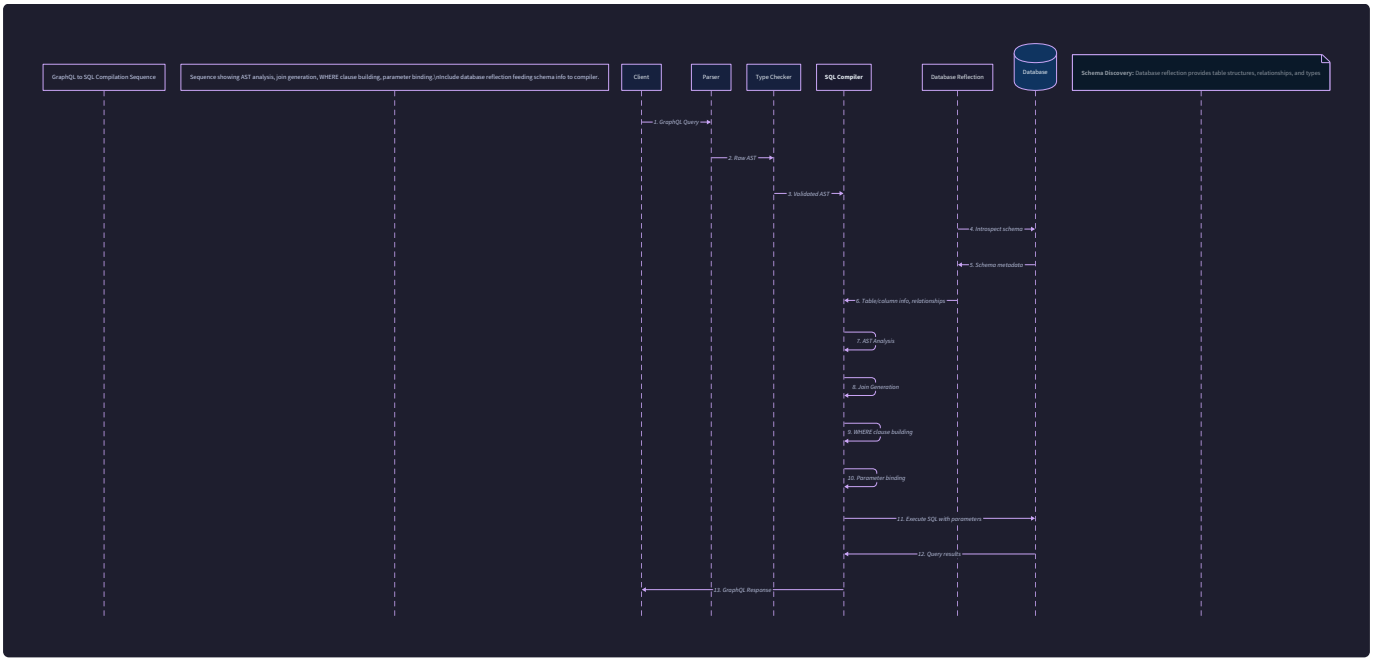
Step 4: Query Validation (Integrated with Execution) Before execution, the engine performs validation using the type system (Milestone 2). This happens implicitly during the execution preparation phase:

1. The operation name (if provided) is used to select the correct `OperationDefinition` from the `Document`.
2. Variable values are coerced to their defined types using `VariableDefinition` type references.
3. The selection set is validated against the schema: fields must exist, arguments must be of correct type, fragments must be applicable. Validation failures produce `GraphQLError` objects and abort execution.

Step 5: SQL Compilation (Milestone 5) For queries against a reflected database schema, the `compile_to_sql` function transforms the validated operation into executable SQL:

1. **Analysis**: The compiler analyzes the `Document` AST, identifying the root table from the query type and tracing nested selections through relationships.
2. **Join Planning**: The `plan_joins` function determines the optimal JOIN order and type based on foreign key relationships in the `DatabaseMetadata`.
3. **SQL Generation**: The compiler builds a `SQLQuery` object containing `SQLSelect` structures with JOINs, WHERE conditions from arguments, and ORDER BY/LIMIT from pagination arguments.
4. **Parameter Binding**: All GraphQL arguments are converted to `SQLParameter` objects to prevent SQL injection.

This step produces a complete, parameterized SQL query ready for execution. You can see the detailed sequence in



.

Step 6: Database Execution The engine executes the generated SQL against the database connection:

1. Parameter values are bound to the SQL placeholders.
2. The query is sent to the database.
3. Raw rows are returned as a list of dictionaries (or similar record format).

Step 7: Result Transformation and Field Resolution The execution engine (Milestone 3) processes the results:

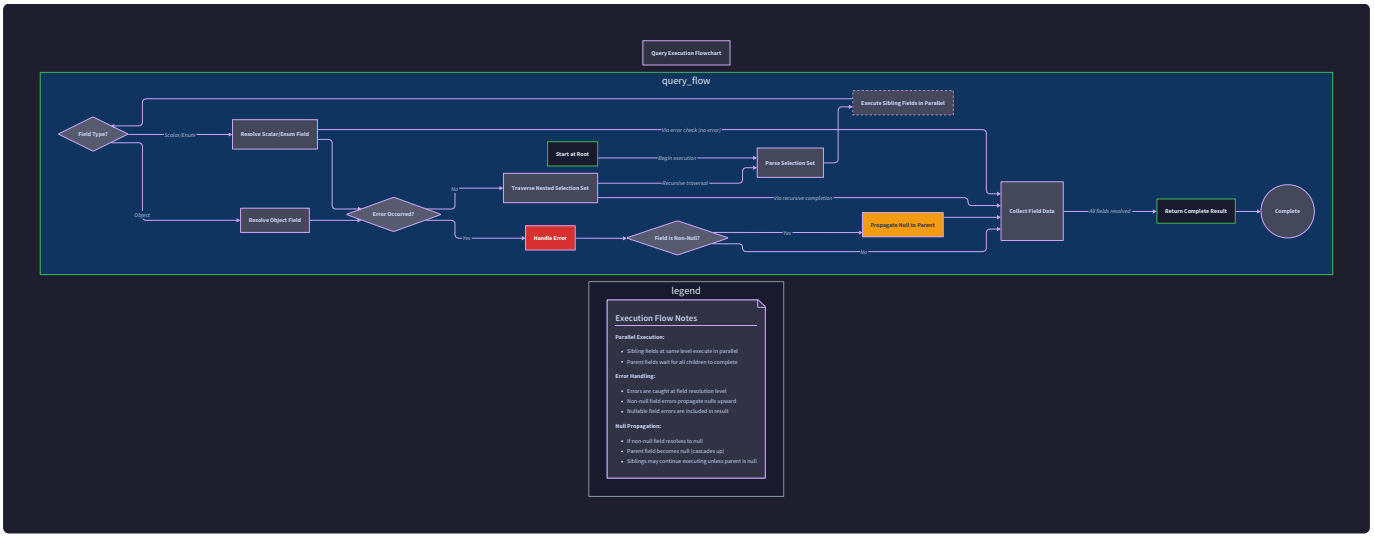
1. **Root Resolution**: For reflected queries, the root resolver uses a `result_mapper` function from the `SQLQuery` to transform flat SQL rows into nested structures matching the GraphQL selection shape.
2. **Nested Field Resolution**: For fields not satisfied by the SQL JOINs (or for custom resolvers), the engine invokes resolver functions from the `GraphQLField` definitions.
3. **Parallel Execution**: Sibling fields at the same level are resolved concurrently when independent.

4. **Error Handling**: Field-level errors are captured as `GraphQLError` objects without aborting the entire query.
5. **Null Propagation**: If a non-null field resolves to null or raises an error, the engine propagates null to the nearest nullable parent per GraphQL specification.

Step 8: Response Assembly The execution engine assembles the final `ExecutionResult`:

1. The `data` field contains the fully resolved response tree.
2. The `errors` field contains any `GraphQLError` objects collected during execution.
3. The result is serialized to JSON by the HTTP layer and returned to the client.

The entire execution flow, showing resolver traversal and parallel branches, is depicted in



.

> **Architecture Decision: Integrated vs. Separate Validation Phase**
>
> - **Context**: GraphQL execution requires validation against the schema, but we must decide when this happens relative to parsing and execution.
> - **Options Considered**:
>   1. **Separate validation phase**: Parse → Validate → Execute (clear separation, catches errors early)
>   2. **Integrated validation**: Parse → Execute with validation as first step (simpler control flow)
>   3. **Lazy validation**: Validate only the parts of the query as they're executed (optimistic, complex error handling)
> - **Decision**: Integrated validation (option 2)
> - **Rationale**: The GraphQL spec defines validation as occurring before execution begins, not interleaved. A separate phase would require an additional traversal of the AST. By integrating validation as the first step of execution, we maintain clear separation conceptually while avoiding duplicate AST traversal. This aligns with reference implementations like graphql-js.
> - **Consequences**: Validation errors prevent any resolver execution, which is correct spec behavior. The execution engine must handle validation errors as a special case before beginning resolver dispatch.

Option	Pros	Cons	Chosen?
Separate validation phase	Clear separation of concerns, easier to test validation independently	Extra AST traversal overhead, more complex control flow	
Integrated validation	**Single AST traversal, simpler control flow, matches reference implementations**	**Validation and execution logic coupled in same component**	✓
Lazy validation	Could avoid validating unused fragments, potential performance gain	Complex error handling, violates spec requirement that validation occurs before execution	

Schema Building and Reflection Flow

Schema building can occur through two primary paths: static programmatic definition or dynamic reflection from a database. The reflection flow (Milestone 4) integrates deeply with the type system (Milestone 2) to auto-generate a complete GraphQL schema.

> **Mental Model: Archaeological Excavation**
> Imagine discovering an ancient city buried underground. The reflection process is like the archaeological dig: you first survey the site (connect to database), then carefully uncover each structure (introspect tables), map the pathways between buildings (detect relationships), and finally produce a detailed model of the city (GraphQL schema) that people can explore without digging themselves.

The reflection process follows a structured algorithm:

Step 1: Connection and Metadata Extraction The `reflect_schema` function is called with database connection configuration and `ReflectionOptions`. It:

1. Establishes a connection to the database using a `BaseIntrospector` adapter.
2. Calls `get_database_metadata()` to retrieve `DatabaseMetadata` containing `TableMetadata` for each table and `ColumnMetadata` for each column.
3. Optionally filters tables based on `include_tables` / `exclude_tables` patterns.

Step 2: Type Mapping For each table, the reflection engine:

1. Creates an `ObjectType` with a name derived from the table name using the configured `NamingConvention`.
2. For each column in `TableMetadata.columns`, uses `TypeMapper.map_column_type()` to convert the SQL data type to a `GraphQLType` (wrapped with `NonNullType` if `is_nullable` is false).
3. Adds a `GraphQLField` for each column to the object type's fields dictionary.

Step 3: Relationship Detection The `RelationshipAnalyzer.analyze()` method processes foreign key constraints from the metadata:

1. Identifies all foreign keys between tables.
2. Determines cardinality (one-to-one, one-to-many) by checking uniqueness constraints.
3. Creates `Relationship` objects linking source and target tables.
4. For each relationship, adds appropriate fields to the GraphQL types:
   - For one-to-many: adds a plural field (e.g., `posts`) to the "one" side returning a list of the "many" type
   - For many-to-one: adds a singular field (e.g., `author`) to the "many" side returning the "one" type

Step 4: Root Query Type Construction The reflection engine builds the root `Query` type:

1. For each table, adds a field to fetch a single record by primary key (e.g., `user(id: ID!)`).
2. For each table, adds a plural field to fetch all records with optional filtering, pagination, and ordering (e.g., `users(where: UserFilter, limit: Int)`).
3. Adds connection fields for Relay-style cursor pagination if configured.

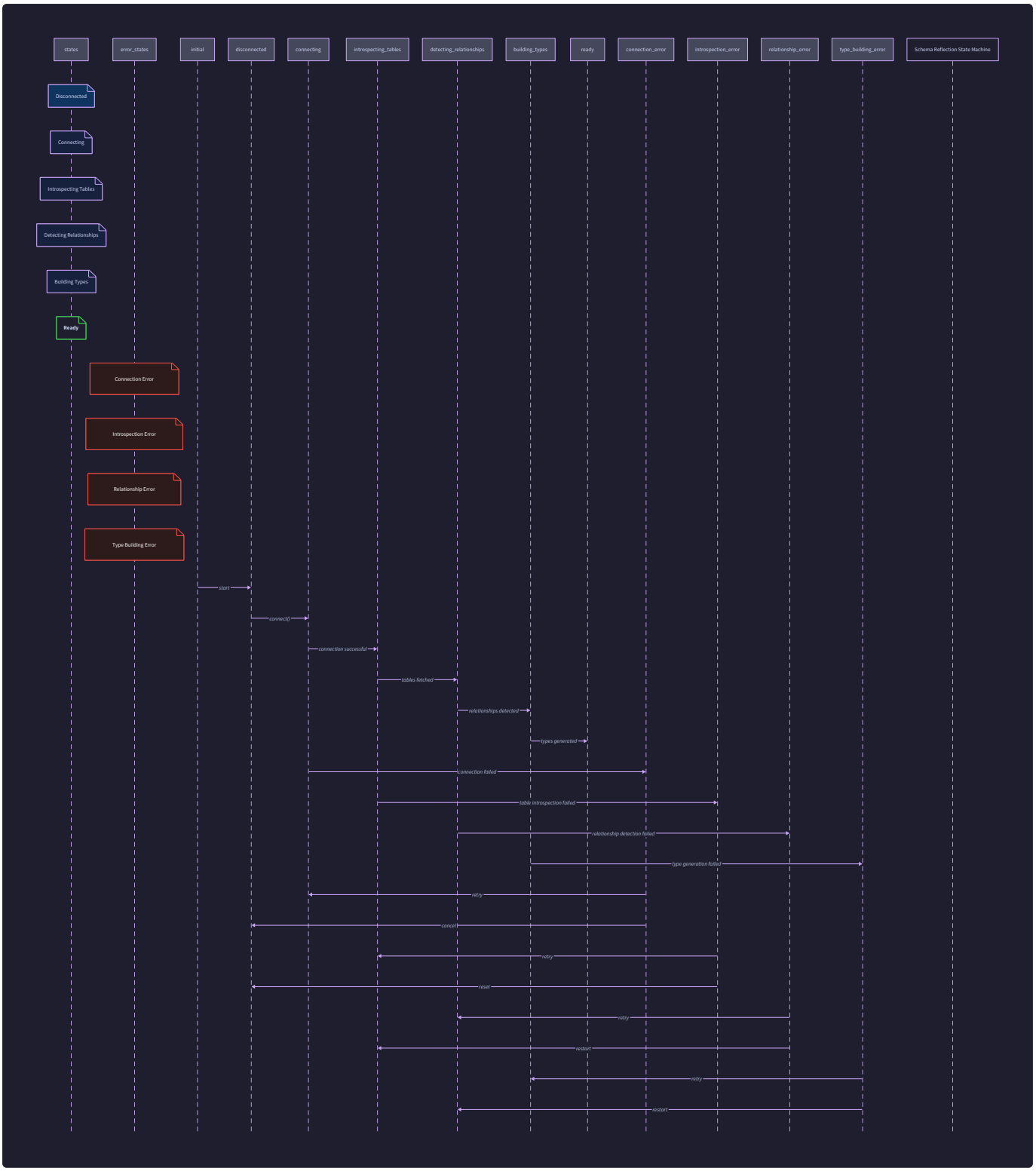
Step 5: Mutation Type Construction (Optional) If `readonly` option is false, the engine builds a `Mutation` type:

1. For each table, adds `create`, `update`, and `delete` mutations with appropriate input types.
2. Input types (`InputObjectType`) are created from table columns, excluding computed or read-only columns.

Step 6: Schema Assembly and Validation The final step uses `create_schema()` to assemble the reflected types into a complete `Schema`:

1. All generated types are collected in the `types` dictionary.
2. The root `Query` and `Mutation` types are set.
3. `validate_schema()` is called to ensure consistency (no circular references, valid interface implementations, etc.).

Step 7: Integration with Execution Engine The generated `Schema` is now indistinguishable from a manually defined one and can be used directly by the query execution engine. The reflection state transitions are shown in


states
error_states
initial
disconnected
connecting
introspecting_tables
detecting_relationships
building_types
ready
connection_error
introspection_error
relationship_error
type_building_error
Schema Reflection State Machine
Disconnected
Connecting
Introspecting Tables
Detecting Relationships
Building Types
Ready
Connection Error
Introspection Error
Relationship Error
Type Building Error
start
connect()
connection successful
tables fetched
relationships detected
types generated
connection failed
table introspection failed
relationship detection failed
type generation failed
retry
cancel
retry
reset
retry
restart
retry
restart

.

Reflection Stage	Input	Output	Key Methods
Connection	Connection string, `ReflectionOptions`	Database connection	`BaseIntrospector.connect()`
Metadata extraction	Database connection	`DatabaseMetadata`	`get_database_metadata()`
Type mapping	`TableMetadata`, `ColumnMetadata`	`ObjectType` with field mappings	`TypeMapper.map_column_type()`
Relationship detection	`DatabaseMetadata`	`Dict[str, List[Relationship]]`	`RelationshipAnalyzer.analyze()`
Root type construction	All generated object types	`ObjectType` for Query and Mutation	Custom generation logic
Schema assembly	All generated types	`Schema`	`create_schema()`, `validate_schema()`

Architecture Decision: Static vs. Dynamic Reflection

- **Context**: We must decide when schema reflection occurs: at startup (static) or on-demand per request (dynamic).
- **Options Considered**:
  1. **Static reflection**: Reflect once at server startup, cache the schema
  2. **Dynamic reflection**: Reflect on every request or schema change
  3. **Hybrid approach**: Reflect at startup but watch for schema changes via database event notifications
- **Decision**: Static reflection with optional reload triggers
- **Rationale**: Database schemas change infrequently in production, and reflection requires expensive queries to information_schema. Static reflection provides predictable performance and allows schema validation at startup. We include a manual reload trigger for schema updates. This balances performance with flexibility.
- **Consequences**: Schema changes require server restart or manual reload. The server must handle connection pooling separately from reflection. This approach is used by both Hasura and PostGraphile in their default configurations.

Option	Pros	Cons	Chosen?
Static reflection	**Fast query execution, startup validation catches errors early**	**Schema changes require restart/reload**	✓
Dynamic reflection	Always current with database, no restart needed	Performance overhead on every request, complex caching needed	
Hybrid approach	Best of both worlds when database supports events	Complex to implement, database-specific event systems	

Message and Data Formats

Components communicate through well-defined data structures. These structures flow through the system, transforming at each stage. Understanding their exact shape is crucial for implementing the interfaces correctly.

AST Nodes (Parser to Type System/Executor)

The parser produces a `Document` containing `Definition` nodes. These are read-only data structures passed to subsequent components.

Structure	Contained In	Purpose	Key Fields
`Document`	Output of `parse_query()`	Root container for all definitions in a GraphQL document	`definitions: List[Definition]`
`OperationDefinition`	`Document.definitions`	A query, mutation, or subscription operation	`operation_type: str`, `selection_set: SelectionSet`, `variable_definitions: List[VariableDefinition]`
`SelectionSet`	`OperationDefinition`, `Field`	A set of fields/fragments to be selected	`selections: List[Selection]`
`Field`	`SelectionSet.selections`	A single field selection with arguments and sub-selections	`name: str`, `arguments: List[Argument]`, `selection_set: Optional[SelectionSet]`
`FragmentDefinition`	`Document.definitions`	A reusable fragment definition	`name: str`, `type_condition: NamedType`, `selection_set: SelectionSet`
`FragmentSpread`	`SelectionSet.selections`	A reference to a named fragment	`name: str`, `directives: List[Directive]`
`VariableDefinition`	`OperationDefinition.variable_definitions`	Definition of a variable used in the operation	`variable: Variable`, `type: Type`, `default_value: Optional[Value]`

Type System Structures (Schema Building)

The type system components create and use these structures to represent the GraphQL schema:

Structure	Used By	Purpose	Key Fields
`Schema`	All components	Complete GraphQL schema definition	`query_type: ObjectType`, `types: Dict[str, GraphQLType]`, `directives: Dict[str, GraphQLDirective]`
`ObjectType`	Schema, Reflection	GraphQL object type with fields	`fields: Dict[str, GraphQLField]`, `interfaces: List[InterfaceType]`
`GraphQLField`	`ObjectType.fields`	Definition of a single field on a type	`type: GraphQLType`, `args: Dict[str, GraphQLArgument]`, `resolve: Optional[Callable]`
`InterfaceType`	Schema	GraphQL interface type	`fields: Dict[str, GraphQLField]`, `resolve_type: Optional[Callable]`
`UnionType`	Schema	GraphQL union type	`types: List[ObjectType]`, `resolve_type: Optional[Callable]`
`InputObjectType`	Schema	Input object type for arguments	`fields: Dict[str, GraphQLInputField]`

Execution State (Executor Internal)

The execution engine maintains state throughout query resolution:

Structure	Lifetime	Purpose	Key Fields
`ExecutionContext`	Per query execution	Tracks all state during execution	`schema: Schema`, `document_ast: Document`, `variable_values: Dict[str, Any]`, `errors: List[GraphQLError]`, `data_loaders: Dict[str, DataLoader]`, `path: List[PathSegment]`
`GraphQLError`	Error occurrence to response	Represents an error in query execution	`message: str`, `locations: List[Location]`, `path: List[PathSegment]`, `extensions: Dict[str, Any]`
`DataLoader`	Per request or context	Batches and caches data fetching calls	`batch_load_fn: Callable`, `cache: Dict`, `queue: List`, `max_batch_size: int`
`ExecutionResult`	End of execution	Final result returned to caller	`data: Optional[Dict]`, `errors: List[GraphQLError]`, `extensions: Dict[str, Any]`

SQL Intermediate Representation (Compiler to Database)

The SQL compiler produces intermediate structures that bridge GraphQL and SQL:

Structure	Produced By	Consumed By	Purpose
`SQLQuery`	`compile_to_sql()`	Database executor	Complete executable SQL query with metadata
`SQLSelect`	SQL generation	`SQLQuery`	A single SELECT statement component
`SQLJoin`	Join planning	`SQLSelect.joins`	A JOIN between two tables
`SQLExpression`	WHERE/ON clause generation	`SQLSelect.where`, `SQLJoin.on`	A condition expression
`SQLParameter`	Parameter binding	Database execution	A bound parameter value

Database Metadata (Reflection Internal)

The reflection engine uses these structures to represent database schema information:

Structure	Source	Purpose	Key Fields
`DatabaseMetadata`	`get_database_metadata()`	Complete database schema snapshot	`tables: Dict[str, TableMetadata]`, `enums: Dict[str, List[str]]`, `version: str`
`TableMetadata`	Database introspection	Metadata for a single table	`name: str`, `schema: str`, `columns: List[ColumnMetadata]`, `primary_key_columns: List[str]`, `foreign_keys: List[Dict[str, Any]]`
`ColumnMetadata`	Database introspection	Metadata for a single column	`name: str`, `data_type: str`, `is_nullable: bool`, `is_primary_key: bool`, `default_value: Optional[str]`
`Relationship`	`RelationshipAnalyzer.analyze()`	A relationship between two tables	`source_table: str`, `target_table: str`, `relationship_type: str`, `source_columns: List[str]`, `target_columns: List[str]`

> **Key Insight: The SQL IR as a Bridge**
> The `SQLQuery` structure is the critical bridge between the GraphQL and SQL worlds. It contains both the executable SQL (as strings and parameters) and the metadata needed to transform flat SQL rows back into nested GraphQL responses. The `result_mapper` function is generated during compilation and understands how to reconstruct the hierarchical response from the denormalized JOIN results.

Data Transformation Pipeline

The complete transformation pipeline shows how data evolves through the system:

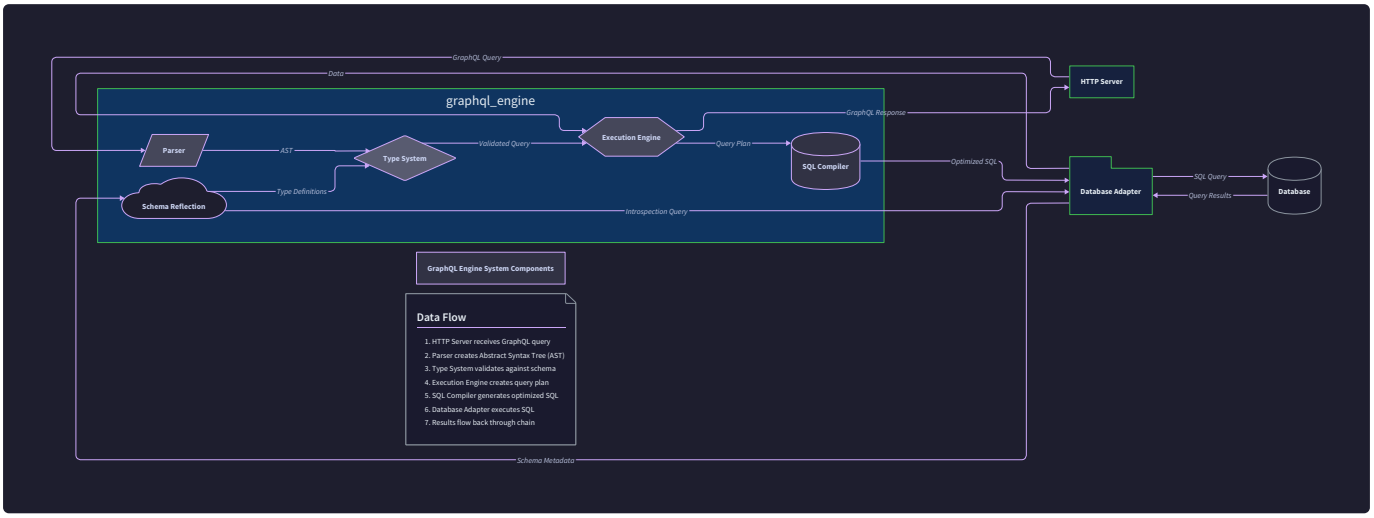
Stage	Input	Output	Transformation
1. Lexing	GraphQL query string	`List[Token]`	Character grouping into lexical tokens
2. Parsing	`List[Token]`	`Document` AST	Recursive descent parsing builds tree structure
3. Validation	`Document` + `Schema`	Validated `Document`	Type checking, field existence, argument validation
4. SQL Compilation	`Document` + `Schema` + `DatabaseMetadata`	`SQLQuery`	AST analysis, join planning, SQL generation
5. Database Execution	`SQLQuery`	`List[Dict]` (rows)	Parameter binding, SQL execution, row fetching
6. Result Mapping	`List[Dict]` + `SQLQuery.result_mapper`	Nested data dict	Row denormalization, nesting reconstruction
7. Field Resolution	Nested data dict + `Schema`	Complete response tree	Resolver invocation, error collection, null propagation
8. Response Assembly	Response tree + errors	`ExecutionResult`	Formatting, error inclusion, extension addition

Component Interface Summary

The primary interfaces between components are defined by these key methods:

Method	From Component	To Component	Purpose
`parse_query(query_str)`	HTTP layer/executor	Parser	Convert query string to AST
`create_schema(...)`	Application/reflection	Type system	Build validated schema from type definitions
`reflect_schema(connection_config, options)`	Application	Reflection engine	Generate schema from database
`execute_query(schema, document_ast, ...)`	HTTP layer	Execution engine	Execute query and return result
`compile_to_sql(schema, document_ast, ...)`	Execution engine	SQL compiler	Generate SQL from GraphQL query
`get_database_metadata(...)`	Reflection engine	Database introspector	Extract raw database schema information

The component interactions are visualized in



, showing how data flows between these interfaces.

Implementation Guidance

This section provides practical guidance for implementing the data flows and interactions described above. We'll focus on wiring the components together and creating the necessary integration points.

Technology Recommendations

Integration Point	Simple Option	Advanced Option
HTTP Server	Flask (Python) with synchronous execution	FastAPI (Python) with async/await support
Database Connection Pooling	`psycopg2.pool.SimpleConnectionPool` (PostgreSQL)	`asyncpg` with built-in connection pool
SQL Building	String concatenation with parameter placeholders	SQLAlchemy Core for dialect-aware SQL generation
Result Caching	`functools.lru_cache` for schema reflection	Redis for distributed query result caching

Recommended File/Module Structure

To organize the interactions between components, create this directory structure:

```
graphql_engine/
├── __init__.py
├── server.py                    # HTTP server integration
├── parser/                      # Milestone 1
│   ├── __init__.py
│   ├── lexer.py                 # Tokenizer
│   ├── parser.py                # Parser, parse_query()
│   └── ast.py                   # AST node classes
├── type_system/                 # Milestone 2
│   ├── __init__.py
│   ├── schema.py                # Schema, create_schema()
│   ├── types.py                 # GraphQLType hierarchy
│   ├── validation.py            # validate_schema()
│   └── introspection.py         # Introspection query handling
├── execution/                   # Milestone 3
│   ├── __init__.py
│   ├── executor.py              # execute_query(), ExecutionContext
│   ├── resolvers.py             # Default resolvers
│   ├── errors.py                # GraphQLError
│   └── dataloaders.py           # DataLoader implementation
├── reflection/                  # Milestone 4
│   ├── __init__.py
│   ├── reflector.py             # reflect_schema()
│   ├── introspectors.py         # BaseIntrospector implementations
│   ├── type_mapper.py           # TypeMapper
│   ├── relationship_analyzer.py # RelationshipAnalyzer
│   └── naming.py                # NamingConvention utilities
├── compilation/                 # Milestone 5
│   ├── __init__.py
│   ├── compiler.py              # compile_to_sql()
│   ├── sql_builder.py           # SQLBuilder
│   ├── join_planner.py          # plan_joins()
│   └── ir.py                    # SQL IR classes
└── integration.py               # Main integration point combining all components
```

Infrastructure Starter Code

Here's a complete HTTP server integration that wires the components together:

PYTHON

```python
# graphql_engine/server.py

from flask import Flask, request, jsonify

from graphql_engine.integration import GraphQLEngine

import logging


app = Flask(__name__)

engine = None


def create_app(database_url=None, schema=None):

    """Create and configure the Flask application with GraphQL engine."""

    global engine



    # Initialize the GraphQL engine

    engine = GraphQLEngine()



    if database_url:

        # Use database reflection for schema

        engine.reflect_schema_from_database(database_url)

    elif schema:

        # Use provided schema

        engine.set_schema(schema)

    else:

        # Use a minimal default schema for testing

        engine.set_default_schema()



    return app


@app.route('/graphql', methods=['POST'])

def graphql_endpoint():

    """Main GraphQL endpoint."""

    if engine is None:

        return jsonify({'errors': [{'message': 'Engine not initialized'}]}), 500



    try:

        # Extract GraphQL request from JSON body

        data = request.get_json()

        query = data.get('query')

        variables = data.get('variables') or {}
```

```
        operation_name = data.get('operationName')


        if not query:
            return jsonify({'errors': [{'message': 'No query provided'}]}), 400


        # Execute the query
        result = engine.execute(query, variables, operation_name)


        # Return the result as JSON
        return jsonify(result)


    except Exception as e:
        logging.exception("GraphQL execution error")
        return jsonify({
            'errors': [{'message': str(e)}]
        }), 500

@app.route('/health', methods=['GET'])
def health_check():
    """Health check endpoint."""
    return jsonify({'status': 'healthy'}), 200

if __name__ == '__main__':
    # For development: create app with database connection
    import os
    database_url = os.environ.get('DATABASE_URL', 'postgresql://localhost/test')
    app = create_app(database_url)
    app.run(debug=True)
```

And the main integration class that orchestrates component interactions:

PYTHON

```python
# graphql_engine/integration.py

from typing import Optional, Dict, Any

from graphql_engine.parser import parse_query

from graphql_engine.type_system.schema import Schema, create_schema

from graphql_engine.execution.executor import execute_query

from graphql_engine.reflection.reflector import reflect_schema

from graphql_engine.compilation.compiler import compile_to_sql


class GraphQLEngine:

    """Main integration point for all GraphQL engine components."""


    def __init__(self):

        self.schema: Optional[Schema] = None

        self.database_connection = None

        self.compilation_enabled = False


    def reflect_schema_from_database(self, connection_string: str,

                                     options: Optional[Dict] = None):

        """Reflect schema from database and enable SQL compilation."""

        from graphql_engine.reflection.reflector import reflect_schema


        # Store connection for later use by compiler

        self.database_connection = self._create_connection(connection_string)


        # Reflect schema from database

        reflection_options = options or {}

        self.schema = reflect_schema(self.database_connection, reflection_options)


        # Enable SQL compilation for this schema

        self.compilation_enabled = True


        return self.schema


    def set_schema(self, schema: Schema):

        """Use a pre-defined schema (disables SQL compilation)."""

        self.schema = schema

        self.compilation_enabled = False
```

```
def set_default_schema(self):
    """Create a minimal default schema for testing."""
    # Implementation creates a simple schema with a hello field
    pass

def execute(self, query: str, variables: Optional[Dict] = None,
            operation_name: Optional[str] = None) -> Dict[str, Any]:
    """Execute a GraphQL query end-to-end."""
    if not self.schema:
        raise ValueError("No schema configured")

    # Step 1: Parse the query
    document_ast = parse_query(query)

    # Step 2: Compile to SQL if enabled
    sql_query = None
    if self.compilation_enabled and self.database_connection:
        sql_query = compile_to_sql(
            self.schema,
            document_ast,
            operation_name,
            variables or {},
            {'connection': self.database_connection}
        )

    # Step 3: Execute the query
    result = execute_query(
        schema=self.schema,
        document_ast=document_ast,
        variable_values=variables,
        operation_name=operation_name,
        # Pass SQL query to execution context for use by resolvers
        context_value={'sql_query': sql_query} if sql_query else None
    )
```

```
        return {
            'data': result.data,
            'errors': [self._format_error(e) for e in result.errors] if result.errors else None
        }

    def _create_connection(self, connection_string: str):
        """Create a database connection based on the connection string."""
        # Simplified implementation - in reality, use proper connection pooling
        import psycopg2
        return psycopg2.connect(connection_string)

    def _format_error(self, error):
        """Format a GraphQLError for JSON response."""
        return {
            'message': error.message,
            'locations': [{'line': loc.line, 'column': loc.column}
                          for loc in error.locations] if error.locations else None,
            'path': error.path
        }
```

Core Logic Skeleton Code

Here's the skeleton for the main execution flow integration:

PYTHON

```python
# graphql_engine/execution/executor.py

from typing import Optional, Dict, Any

from graphql_engine.parser.ast import Document

from graphql_engine.type_system.schema import Schema

from .errors import GraphQLError

from .execution_context import ExecutionContext


class ExecutionResult:

    """Result of executing a GraphQL query."""

    def __init__(self, data: Optional[Dict] = None, errors: Optional[List[GraphQLError]] = None):

        self.data = data

        self.errors = errors or []


def execute_query(

    schema: Schema,

    document_ast: Document,

    variable_values: Optional[Dict[str, Any]] = None,

    operation_name: Optional[str] = None,

    context_value: Any = None,

    root_value: Any = None

) -> ExecutionResult:

    """

    Execute a GraphQL query against a schema.



    This is the main entry point for query execution, integrating parsing,

    validation, and field resolution.



    Args:

        schema: The GraphQL schema to execute against

        document_ast: Parsed AST of the GraphQL query

        variable_values: Values for variables defined in the query

        operation_name: Name of the operation to execute if document contains multiple

        context_value: Context object passed to all resolvers

        root_value: Value passed to the root resolver



    Returns:

        ExecutionResult containing data and/or errors
```

```
    """
    # TODO 1: Validate that schema is properly configured (query_type exists)
    # TODO 2: Extract the operation to execute from document_ast
    #   - If operation_name is provided, find the operation with that name
    #   - If only one operation exists and no name provided, use that one
    #   - Otherwise, raise an error about ambiguous operation
    # TODO 3: Validate variable values against variable definitions
    #   - Coerce values to the defined types
    #   - Apply default values where variables are not provided
    # TODO 4: Create ExecutionContext
    #   - Store schema, document_ast, variable_values, context_value, root_value
    #   - Initialize empty errors list and data_loaders dict
    # TODO 5: Validate the operation against the schema
    #   - Check that all selected fields exist on their respective types
    #   - Validate argument types match field definitions
    #   - Verify fragments are used on compatible types
    #   - If validation fails, collect errors and return early
    # TODO 6: Execute the operation based on its type (query/mutation/subscription)
    #   - For queries: execute selection set starting from root_value
    #   - For mutations: execute serially (not in parallel)
    # TODO 7: Build and return ExecutionResult with data and errors


    # For now, return a placeholder
    return ExecutionResult(data=None, errors=[])
```

And for the SQL compilation integration:

PYTHON

```python
# graphql_engine/compilation/compiler.py

from typing import Dict, Any, Optional

from graphql_engine.parser.ast import Document

from graphql_engine.type_system.schema import Schema

from .ir import SQLQuery


def compile_to_sql(

    schema: Schema,

    document_ast: Document,

    operation_name: Optional[str],

    variable_values: Dict[str, Any],

    context: Dict[str, Any]

) -> SQLQuery:

    """

    Compile a GraphQL query to an optimized SQL query.



    This is the main entry point for SQL compilation, integrating AST analysis,

    join planning, and SQL generation.



    Args:

        schema: The GraphQL schema (must be a reflected schema)

        document_ast: Parsed AST of the GraphQL query

        operation_name: Name of the operation to compile

        variable_values: Values for variables used in the query

        context: Execution context containing database connection and metadata



    Returns:

        SQLQuery object containing executable SQL and parameters

    """

    # TODO 1: Verify the schema is a reflected schema (has database metadata)

    # TODO 2: Extract the operation to compile (similar to execute_query)

    # TODO 3: Analyze the selection set to identify accessed tables and relationships

    #   - Start from the root query type field (e.g., 'users', 'posts')

    #   - Recursively traverse nested selections, mapping them to table relationships

    #   - Build a list of selection paths: e.g., ['users', 'users.posts', 'users.posts.comments']

    # TODO 4: Plan JOINs using plan_joins() function

    #   - Determine optimal join order to avoid cartesian products
```

```
    #   - Choose join types (INNER, LEFT) based on nullability
    #   - Generate table aliases for self-joins or multiple joins to same table
    # TODO 5: Generate SELECT clause for each table
    #   - Include only columns corresponding to selected fields
    #   - Generate column aliases that map back to field names
    # TODO 6: Generate WHERE clause from GraphQL arguments
    #   - Convert filter arguments to SQL conditions
    #   - Handle comparison operators (eq, lt, gt, in, etc.)
    #   - Bind parameters to prevent SQL injection
    # TODO 7: Generate ORDER BY and LIMIT/OFFSET from pagination arguments
    # TODO 8: Build SQLQuery object with all components
    #   - Set root_select with main SELECT statement
    #   - Add nested_selects for complex cases requiring subqueries
    #   - Collect all parameters for binding
    #   - Generate result_mapper function that transforms flat rows to nested structure
    # TODO 9: Return the complete SQLQuery object


    # For now, return a placeholder
    return SQLQuery(root_select=None, nested_selects=[], parameters=[])
```

Language-Specific Hints

Python-Specific Implementation Notes:

1. **Type Hints**: Use Python's type hints extensively for the data structures. This provides documentation and helps catch errors early:

```python
from typing import List, Optional, Dict, Any, Union


class Field(Node):
    name: str
    alias: Optional[str]
    arguments: List[Argument]
    directives: List[Directive]
    selection_set: Optional[SelectionSet]
```

2. **Dataclasses**: Use `@dataclass` decorator for immutable data structures like AST nodes and metadata objects:

```python
from dataclasses import dataclass


@dataclass(frozen=True)
class Location:
    line: int
    column: int
```

3. **Async/Await**: For parallel field execution, use `asyncio.gather()` to resolve sibling fields concurrently:

```python
import asyncio


async def resolve_fields_concurrently(fields):
    tasks = [resolve_field(field) for field in fields]
    return await asyncio.gather(*tasks, return_exceptions=True)
```

4. **Connection Pooling**: Use context managers for database connections to ensure proper cleanup:

```python
from contextlib import contextmanager


@contextmanager
def get_connection():
    conn = pool.getconn()
    try:
        yield conn
    finally:
        pool.putconn(conn)
```

Milestone Checkpoint

After implementing the integration layer, verify the complete flow works:

1. **Start the server**: Run `python -m graphql_engine.server` and ensure it starts without errors.
2. **Test with a simple query**: Use curl or a GraphQL client to send a query:

```bash
curl -X POST http://localhost:5000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "query { __schema { types { name } } }"}'
```

Expected output should include introspection data with no errors.

3. **Test database reflection**: Set a `DATABASE_URL` environment variable and restart. Send a query for a table that exists:

```bash
curl -X POST http://localhost:5000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "query { users { id, name } }"}'
```

Expected: Returns actual user data from the database or an empty array if table is empty.

4. **Verify SQL generation**: Enable debug logging and check that SQL queries are being generated correctly. Look for log messages showing the generated SQL with parameter placeholders.

Signs of Problems:

- "No schema configured" error: The engine wasn't initialized properly
- Parse errors in valid queries: Parser implementation issues
- "Field X doesn't exist" when it does in database: Reflection naming issues
- SQL syntax errors: SQL generation bugs, especially with JOINs
- Missing nested data: Result mapper not reconstructing hierarchy correctly

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Query returns empty data but no error	SQL query returning no rows	Log the generated SQL and execute it manually against the database	Check table names, WHERE conditions, JOIN conditions
Nested fields are null	Result mapper not reconstructing hierarchy	Print the raw SQL rows and the result mapper logic	Ensure result mapper groups child records under parent keys
"Cannot query field X on type Y"	Reflection didn't create the field	Check if table/column was excluded by filters, or naming convention mismatch	Adjust reflection options, check naming conventions
Database connection errors	Connection pool exhausted or credentials wrong	Check connection string, database permissions	Ensure connection pool size adequate, credentials correct
Cartesian product (too many rows)	Missing JOIN conditions	Examine generated SQL JOIN clauses	Ensure every JOIN has an ON condition from foreign key
Memory usage high with large queries	No LIMIT clause on root query	Check if pagination arguments are being applied	Add default limit for collection fields, implement cursor pagination
"Variable $var not defined"	Variable parsing/validation issue	Check variable definitions in operation header	Ensure variable definitions are parsed and variables are passed

To debug effectively:

1. Enable detailed logging for all components
2. Use the GraphiQL interface (if implemented) to see query validation
3. Test SQL queries directly against the database
4. Add `print()` statements in the result mapper to see transformation steps
5. Use a visualizer for AST structure to understand query shape

Error Handling and Edge Cases

> **Milestone(s):** This section spans all five milestones, addressing failure modes and special cases that arise throughout the entire GraphQL execution pipeline. Error handling is a critical cross-cutting concern that must be addressed consistently across parsing (Milestone 1), type validation (Milestone 2), execution (Milestone 3), database reflection (Milestone 4), and SQL compilation (Milestone 5).

Error Categories and Handling

Think of error handling in a GraphQL engine as a **multi-layered safety net**—each layer catches different types of failures while allowing the system to continue operating when possible. Unlike traditional REST APIs where a single error often terminates the entire request, GraphQL's partial results model requires careful error propagation that preserves valid data while clearly indicating what went wrong.

The GraphQL specification defines specific error behaviors, but our implementation must also handle infrastructure failures (database connections, network issues) and implementation bugs. Each error category requires distinct detection strategies and recovery approaches.

Error Classification and Handling Strategies

Error Category	Detection Strategy	Recovery Approach	User Visibility
Parse Errors (Milestone 1)	Lexer/parser token validation, grammar rule violations	Immediate termination with precise location info	Full error details (line, column, expected tokens)
Validation Errors (Milestone 2)	Schema validation passes, query validation against schema	Early rejection before execution	Detailed error messages about schema violations
Execution Errors (Milestone 3)	Resolver exceptions, type coercion failures, null violations	Partial execution with error propagation	Errors attached to specific fields in response
Database Errors (Milestone 4, 5)	SQL execution failures, connection issues, constraint violations	Transaction rollback (if applicable), error categorization	Sanitized error messages (hide DB internals)
Resource Errors (All)	Timeouts, memory limits, connection limits	Graceful degradation, circuit breaking	Generic service unavailable messages

Parse Error Handling

Parse errors occur when the GraphQL query string violates the language syntax. The parser must provide **actionable feedback** by pinpointing exactly where the syntax diverges from expectations.

Decision: Parser Error Reporting

- **Context**: Learners need detailed feedback to debug malformed queries, but overly technical error messages can be confusing.
- **Options Considered**:
  1. Simple "syntax error" messages with line numbers only
  2. Full parser state dumps (current token, expected tokens, parser stack)
  3. Human-readable messages with exact location and suggestions
- **Decision**: Option 3 with precise location tracking and clear expectations
- **Rationale**: Follows GraphQL spec requirements for error locations while being learner-friendly. The `Location` type in our data model provides the necessary tracking.
- **Consequences**: Requires maintaining accurate line/column tracking throughout lexing and parsing, but gives users exactly what they need to fix queries.

Parse Error Type	Detection Point	Error Message Format	Example
Unexpected Token	Parser expecting specific token (e.g., `{` after field name)	`Syntax error at line X, column Y: Expected "{", found "}"`	`query { user(id: 1 }` (missing closing `)`)
Unterminated String	Lexer reaches end of input without closing quote	`Syntax error at line X, column Y: Unterminated string`	`query { name: "unclosed`
Invalid Number	Lexer fails to convert character sequence to number	`Syntax error at line X, column Y: Invalid number "12.3.4"`	`query { value: 12.3.4 }`
Unknown Character	Lexer encounters character outside valid token set	`Syntax error at line X, column Y: Unknown character "@"`	`query { field @unknown }`

The parser must maintain a **parse stack** to provide context-aware error messages. For example, when encountering a syntax error inside a nested selection set, the error should indicate "while parsing selection set of field 'user'" to give context beyond just line/column numbers.

Validation Error Handling

Validation errors occur when a syntactically correct query violates schema rules. These are caught before execution to prevent runtime failures. The validation phase performs dozens of checks defined in the GraphQL specification.

Decision: Validation Strictness

- **Context**: Some validation rules are essential for correctness (like type compatibility), while others are warnings (like deprecated field usage).
- **Options Considered**:
  1. Fail fast on first validation error
  2. Collect all validation errors before reporting
  3. Separate errors from warnings, continue execution with warnings
- **Decision**: Option 2 with all validation errors collected
- **Rationale**: GraphQL spec requires returning all validation errors. This gives users complete information to fix their query without multiple round trips.
- **Consequences**: More complex validation implementation that must continue after encountering errors, but provides better developer experience.

Validation Rule Category	Example Checks	Error Message Pattern
Type Compatibility	Field type matches argument type, return type matches selection	`Field "age" of type "String" cannot represent non-string value: 42`
Field Existence	Requested field exists on type, arguments exist	`Cannot query field "email" on type "User". Did you mean "name"?`
Fragment Validation	Fragment type conditions are valid, spreads are possible	`Fragment "UserFields" cannot be spread here as object of type "Post" can never be of type "User"`
Variable Validation	Variables match defined types, required variables provided	`Variable "$id" of type "Int!" is required but not provided`
Operation Validation	Single operation when unnamed, operation names unique	`This operation is named "GetUser" but no operation name was provided`

Validation should implement the **visitor pattern** to traverse the AST once while running multiple validation rules. Each rule is a separate function that can add errors to a shared collection without stopping the traversal.

Execution Error Handling

Execution errors occur during resolver execution and represent the most complex error handling scenario. GraphQL's **partial results** model means fields can fail independently, and the engine must decide how to propagate these failures.

Mental Model: The Fault-Tolerant Assembly Line Imagine an assembly line where each station (resolver) processes parts of a product (data). If one station fails, we don't throw away the entire product—we mark that specific part as defective and continue assembling the rest. The final product has some missing parts, but we clearly label which ones failed and why.

Execution Error Type	Detection Mechanism	Null Propagation Rule	Error Collection
Resolver Exception	Try/catch around resolver invocation	If field is non-null (`Type!`), propagate to parent	Add to `GraphQLError` list with path
Type Coercion Error	Value validation during argument/result conversion	Follows same rules as resolver errors	Add to `GraphQLError` list with path
Authorization Error	Context validation in resolvers	Typically treat as resolver exception	May include extensions with auth details
Resource Limit Error	Timeout detection, memory monitoring	Terminate entire execution	Single error at root level

The key challenge is **null propagation**: when a non-null field (`String!`) resolves to null or throws an error, the entire parent field becomes null (if nullable) or continues bubbling up. This follows the GraphQL spec's "fail fast" rule for non-null fields.

Execution error handling algorithm:

1. When a resolver throws an exception, catch it immediately
2. Create a `GraphQLError` with the exception message, current field path (`["users", 0, "email"]`), and location from AST
3. If the field's return type is non-null (`Type!`):
   - Set the field's resolved value to `None` (null)
   - Mark the parent field as "contains null from non-null child"
4. If the parent field is also non-null, repeat step 3 recursively
5. If the field is nullable (`Type` or `[Type]`):
   - Set the field's resolved value to `None`
   - Continue executing sibling fields
6. Add the error to `ExecutionContext.errors` list for inclusion in final response

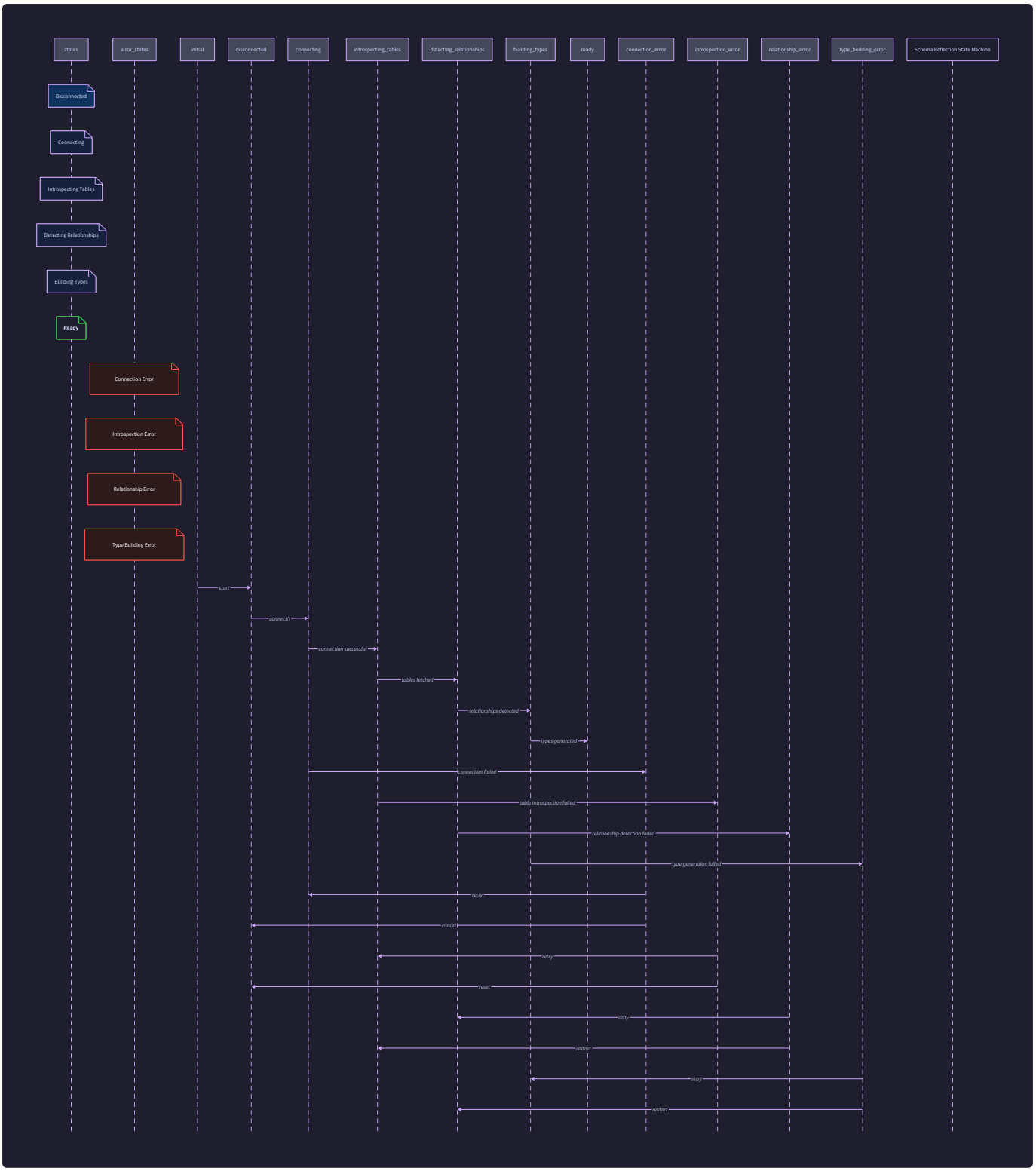
Database Error Handling

Database errors bridge Milestone 4 (reflection) and Milestone 5 (SQL compilation). These errors come from the underlying database system and must be sanitized before exposing to users.

> **Decision: Database Error Sanitization**
>
> - **Context**: Raw database errors expose internal details (table names, constraint names, SQL snippets) that are security risks and confusing to GraphQL API consumers.
> - **Options Considered**:
>   1. Pass through raw database errors (maximum debugging info, maximum risk)
>   2. Completely generic errors ("Database error occurred")
>   3. Categorized errors with sanitized messages
> - **Decision**: Option 3 with error categorization and safe messaging
> - **Rationale**: Balances debuggability (developers need to know what went wrong) with security (don't leak schema details). Different error categories suggest different fixes.
> - **Consequences**: Requires parsing database error messages (which vary by database vendor) and mapping to safe categories.

Database Error Category	Detection Pattern	Sanitized Message	Internal Logging
Connection Errors	Network timeout, authentication failure	"Unable to connect to database"	Full connection string (without password), network details
Constraint Violations	Unique constraint, foreign key violation	"Data validation failed"	Constraint name, table name, violating values
Syntax/Planning Errors	Invalid SQL, missing columns	"Query compilation error"	Generated SQL, compilation context
Permission Errors	Insufficient privileges on table/column	"Access denied to requested resource"	User/role, attempted operation
Resource Errors	Too many connections, disk full	"Database resource limit reached"	Resource type, current usage

The reflection component (Milestone 4) has its own state machine with explicit error states, as shown in the diagram:



Reflection error recovery states:

Current State	Error Event	Next State	Recovery Action
`connecting`	Connection timeout	`error`	Retry with exponential backoff (max 3 attempts)
`introspecting_tables`	Permission denied on information_schema	`partial`	Continue with accessible tables only, log warning
`detecting_relationships`	Circular reference detected	`ready`	Mark relationship as circular, generate appropriate GraphQL types
`building_types`	Type mapping failure	`error`	Fall back to `String` type for problematic column, continue

Error Response Format

All errors eventually serialize to the standard GraphQL error response format:

```json
{
  "data": {
    "user": null,
    "posts": [...]
  },
  "errors": [
    {
      "message": "Cannot return null for non-nullable field User.email",
      "path": ["user", "email"],
      "locations": [{"line": 5, "column": 7}],
      "extensions": {
        "code": "NON_NULL_VIOLATION",
        "timestamp": "2023-10-01T12:34:56Z"
      }
    }
  ]
}
```

The `extensions` field is where implementation-specific details can be added (error codes, timestamps, debug IDs for correlating with server logs).

Null and Optional Value Handling

Null handling in GraphQL is **propagative rather than terminating**—a null value can "bubble up" through the response tree according to specific rules. This behavior fundamentally differs from most programming languages and requires careful implementation.

Mental Model: The Domino Chain Imagine a chain of dominoes where each domino represents a field in your GraphQL response. Non-null fields (`Type!`) are dominoes glued to their neighbors—if one falls (becomes null), it forces the next one to fall too. Nullable fields (`Type`) are unglued dominoes—they can fall independently without affecting others. List fields (`[Type]`) are racks holding multiple domino chains—one chain can fail without toppling the entire rack.

Null Propagation Rules

Type Wrapper	Null Source	Propagation Behavior	Example
Nullable field (`User`)	Field resolver returns `None`	Field becomes `null` in response, siblings execute normally	`{ user(id: 999) { name } }` → `{"data": {"user": null}}`
Non-null field (`User!`)	Field resolver returns `None`	Parent field becomes `null` (if nullable), continues upward	`{ user: user(id: 999) { name } }` → `{"data": {"user": null}}`
List of nullable (`[User]`)	One element resolves to `None`	`null` appears in list at that position	`{ users: [{name: "A"}, null, {name: "C"}] }`
List of non-null (`[User!]`)	One element resolves to `None`	Entire list becomes `null`	`{ users: null }` (if any user is null)
List of non-null in non-null (`[User!]!`)	One element resolves to `None`	Parent field becomes `null`	`{ users: null }` (bubbles up)

The implementation must track **null propagation chains** during execution. When a resolver returns `None`, check its return type:

- If `NonNullType`: find the nearest nullable parent by walking up the type tree
- If nullable: stop propagation, set value to `null`
- If inside a `ListType`: check if list elements are non-null

Database NULL to GraphQL Null Mapping

Database NULL values present special challenges because different database types have different semantics for NULL:

Database NULL Context	GraphQL Representation	Default Behavior	Override Option
Nullable column with NULL	Field returns `null`	Direct mapping	Custom scalar with different null handling
Non-nullable column (NOT NULL constraint)	Field is non-null (`Type!`)	Schema generation marks as non-null	Can be overridden in reflection options
LEFT JOIN missing row	Nested object field returns `null`	Entire related object is `null`	Configuration to return empty object
Aggregate function on empty set	`COUNT` returns 0, `AVG` returns `null`	Follow SQL semantics	Custom resolvers to normalize

The reflection component (Milestone 4) must read `ColumnMetadata.is_nullable` to determine whether to generate nullable (`String`) or non-null (`String!`) GraphQL fields. However, there's a subtlety: a database column might be technically nullable but semantically required (like a `deleted_at` timestamp that's NULL for active records). The `ReflectionOptions` should allow overriding nullability decisions.

Optional Arguments and Default Values

GraphQL arguments can be optional with default values, which creates three possible states for each argument:

Argument State	GraphQL Syntax	Internal Representation	Behavior
Provided with value	`user(id: 5)`	`Argument.value = IntValue(5)`	Use provided value
Provided with null	`user(id: null)`	`Argument.value = NullValue()`	Use explicit null (overrides default)
Not provided	`user`	No argument in argument list	Use default value from schema

The execution engine must distinguish between "argument not provided" and "argument provided as null" because they have different semantics. During argument coercion:

1. Check if argument exists in the query's `arguments` list
2. If missing: use `GraphQLArgument.default_value` from schema
3. If present: coerce the provided value to expected type
4. If value is `NullValue()`: pass `None` to resolver (even if default exists)

This behavior is critical for mutations where `null` might mean "clear this field" while absence means "don't change this field."

Edge Case Scenarios

Edge cases test the robustness of the GraphQL engine and often reveal bugs in less-traveled code paths. These scenarios require special handling beyond the normal execution flow.

Empty Selections and Minimal Queries

An empty selection set might seem pointless, but it's valid GraphQL and must be handled gracefully. This often occurs in automated query generation or as a placeholder.

Empty selection handling algorithm:

1. During query validation: empty selection sets are allowed on object types (returns empty object)
2. During execution: when encountering an empty selection set:
   - Return empty dictionary `{}` for the field
   - Still execute field-level directives if present
   - Still validate field arguments
3. For interfaces/unions: empty selection sets are invalid unless `__typename` is implicitly added

The minimal valid query `{ __typename }` (just the meta-field) should execute successfully and return `{"data": {"__typename": "Query"}}`.

Circular References

Circular references occur in two domains: GraphQL type definitions and database foreign key relationships. Each requires different handling strategies.

GraphQL type circularity:

```graphql
type User {
  friends: [User!]!
}

type Query {
  user: User!
}
```

This is valid and requires the type system to handle recursive type references during:

- Schema validation: must allow recursive types but detect infinite loops in input types
- Query execution: must have depth limits to prevent infinite recursion
- Introspection: must not get stuck in infinite loops when generating type descriptions

Database relationship circularity:

```sql
-- Employee table with manager hierarchy

CREATE TABLE employees (

  id INT PRIMARY KEY,

  name VARCHAR(100),

  manager_id INT REFERENCES employees(id)

);
```

The reflection component must detect self-referential foreign keys and generate appropriate GraphQL types without infinite recursion. The generated schema should include:

- `Employee` type with `manager: Employee` field (nullable)
- Query field with argument to control traversal depth
- Automatic depth limiting in generated resolvers

Circular reference detection algorithm (reflection):

1. Build directed graph of table relationships from foreign keys
2. Run cycle detection (DFS with back edges)
3. For each cycle found:
   - Mark relationships in cycle with `is_self_referential = True`
   - Generate GraphQL types with nullable back-references to break infinite traversal
   - Add `max_depth` argument to collection fields with default limit (e.g., 10)
4. During SQL compilation: apply `JOIN` limits based on query arguments

Deeply Nested Queries

Deeply nested queries can cause performance problems (exponential JOIN explosion) and even stack overflows in recursive execution. The engine must implement **reasonable limits**.

Limit Type	Default Value	Configuration	Enforcement Point
Query Depth	20 levels	`max_query_depth` in execution context	Query validation phase
Selection Width	50 fields per level	`max_fields_per_level`	Execution planning
List Size	1000 items	`max_list_size` in DataLoader	SQL LIMIT clause
JOIN Count	15 tables	`max_joins` in SQL compiler	JOIN planning

When limits are exceeded:

1. During validation: reject query with clear error "Query too deep (max 20 levels)"
2. During execution: truncate results (e.g., only return first 1000 list items)
3. During SQL compilation: fail with suggestion to add more filters

The limits should be configurable per-request via `ExecutionContext` for different use cases (internal API vs. public API).

Large Results and Pagination Edge Cases

GraphQL doesn't have built-in pagination, but our auto-generated schemas include pagination arguments (`first` , `after` , `last` , `before`). These create several edge cases:

Pagination Edge Case	Behavior	Implementation Consideration
Negative `first` / `last`	Validation error	Must validate arguments before SQL generation
Both `first` and `last`	Use `first` (GraphQL best practice)	Document precedence in generated schema comments
`after` cursor for deleted item	Start from next available item	Cursors must be resilient to data deletion
Empty cursor string	Treat as "start from beginning"	Validate cursor format before decoding
`before` without `last`	Ignore `before` or error	Consistent policy needed

Cursor-based pagination implementation must handle:

- **Cursor encoding/decoding**: Base64-encoded JSON with table name, primary key values, sort values
- **Cursor validation**: Ensure cursor is for the correct table and query
- **Cursor stability**: Cursors must remain valid across data updates (use immutable columns or timestamps)

Type System Edge Cases

The GraphQL type system has subtle edge cases that trip up implementations:

Edge Case	GraphQL Spec Rule	Implementation Strategy
Interface field nullability	Implementing field can be more strict (non-null) but not more loose	Validation must check interface implementation contravariantly
Union with overlapping types	Types in union must be distinct object types	Schema validation ensures no type appears twice
Empty enum values	Enum must have at least one value	Schema validation rejects empty enums
Input object default values	Default values must be coercible to field type	Validate defaults during schema building
List of lists (`[[Int]]`)	Valid type, must handle nested list coercion	Recursive type handling in execution

Interface implementation validation algorithm:

1. For each interface field, find corresponding object field
2. Check field type compatibility:
   - If interface field is nullable (`T`), object field can be `T` or `T!`
   - If interface field is non-null (`T!`), object field must be `T!`
   - Covariant for object types: `Dog` can implement `Pet` if `Dog` is subtype
3. Check argument compatibility (contravariant):
   - Object field can omit optional interface arguments
   - Object field arguments must have compatible types (more general)

SQL Compilation Edge Cases

The GraphQL-to-SQL compiler faces unique edge cases when bridging the hierarchical and relational models:

Compilation Edge Case	Problem	Solution
Cartesian product from multiple one-to-many	`{ users { posts tags } }` creates `users × posts × tags`	Use `LATERAL JOIN` or separate queries with batching
Diamond relationships	`{ user { posts author } }` where `posts.author` is same as `user`	Deduplicate JOINs, use table aliases
Polymorphic relationships	Interface/union types with different backing tables	Generate `CASE` statements or multiple queries
Missing JOIN condition	Relationship exists in GraphQL schema but not in database (view)	Fall back to separate query or error
SQL dialect differences	`LIMIT/OFFSET` vs `FETCH NEXT` vs `TOP`	Abstract behind `SQLDialect` enum

Diamond relationship handling algorithm:

1. Detect when same table appears multiple times in join tree
2. Generate unique table aliases (`posts`, `posts_2`)
3. Ensure JOIN conditions use correct aliases
4. In result mapper: map aliased columns back to correct object paths

Polymorphic relationship compilation:

```graphql
query {
  search(query: "graphql") {
    ... on Book { title author }
    ... on Article { title url }
  }
}
```

Compiles to:

```sql
(SELECT 'Book' AS __typename, title, author FROM books WHERE title LIKE ?)

UNION ALL

(SELECT 'Article' AS __typename, title, url FROM articles WHERE title LIKE ?)
```

With the result mapper using `__typename` to route to correct GraphQL types.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Error Tracking	List of error dictionaries in execution context	Structured error tree with severity levels
Null Propagation	Simple recursion with type checking	Memoization of nullability decisions
Database Error Parsing	String matching on error messages	Database-specific error code mapping
Query Limits	Hard-coded constants	Dynamic limits based on query complexity scoring
Cursor Pagination	Offset-based pagination (simple)	Cursor-based with Base64 encoding (spec-compliant)

B. Recommended File/Module Structure

```
graphql-engine/
  errors/
    __init__.py              # Error base classes and utilities
    parse_errors.py          # Parse error definitions (Milestone 1)
    validation_errors.py     # Validation error definitions (Milestone 2)
    execution_errors.py      # Execution error definitions (Milestone 3)
    database_errors.py       # Database error handling (Milestones 4-5)
    null_handling.py         # Null propagation logic (Milestone 3)
    limits.py                # Query depth/width/size limits
  execution/
    context.py               # ExecutionContext with errors list
    executor.py              # Error-aware executor
```

C. Infrastructure Starter Code

Complete Error Base Class:

PYTHON

```python
# errors/__init__.py

from dataclasses import dataclass, field

from typing import List, Optional, Dict, Any, Union

from .types import Location, PathSegment


@dataclass

class GraphQLError(Exception):

    """Base class for all GraphQL errors."""

    message: str

    locations: List[Location] = field(default_factory=list)

    path: List[PathSegment] = field(default_factory=list)

    original_error: Optional[Exception] = None

    extensions: Dict[str, Any] = field(default_factory=dict)



    def to_dict(self) -> Dict[str, Any]:

        """Convert error to GraphQL response format."""

        result = {

            "message": self.message,

            "locations": [{"line": loc.line, "column": loc.column}

                          for loc in self.locations],

        }

        if self.path:

            result["path"] = self.path

        if self.extensions:

            result["extensions"] = self.extensions

        return result


class GraphQLSyntaxError(GraphQLError):

    """Syntax error during parsing."""

    pass


class GraphQLValidationError(GraphQLError):

    """Validation error against schema."""

    pass


class GraphQLExecutionError(GraphQLError):

    """Error during query execution."""

    pass
```

```
class GraphQLNonNullError(GraphQLExecutionError):

    """Special error for non-null field violations."""

    def __init__(self, field_name: str, path: List[PathSegment]):

        super().__init__(

            message=f"Cannot return null for non-nullable field {field_name}",

            path=path,

            extensions={"code": "NON_NULL_VIOLATION"}

        )
```

Complete Null Propagation Helper:

PYTHON

```python
# errors/null_handling.py

from typing import Any, Optional, Tuple

from ..types import GraphQLType, NonNullType, ListType


def should_propagate_null(value: Any, field_type: GraphQLType) -> Tuple[bool, Optional[GraphQLType]]:

    """

    Determine if null should propagate upward.

    Returns (should_propagate, nullable_parent_type).

    """

    if value is not None:

        return (False, None)



    # Check type wrapper chain

    current_type = field_type

    type_stack = []



    # Unwrap type modifiers

    while isinstance(current_type, (NonNullType, ListType)):

        type_stack.append(current_type)

        if isinstance(current_type, NonNullType):

            # Found non-null wrapper

            # If this is the outermost non-null, propagate

            if not type_stack or all(not isinstance(t, NonNullType) for t in type_stack[:-1]):

                return (True, None)

        current_type = current_type.of_type



    return (False, None)


def apply_null_propagation(result: Dict[str, Any], error: GraphQLError) -> Dict[str, Any]:

    """

    Apply null propagation rules to result dictionary.

    """

    if not error.path:

        return result



    # Navigate to the error location in the result

    current = result
```

```
    for i, segment in enumerate(error.path[:-1]):
        if isinstance(segment, str):
            current = current.get(segment, {})
        elif isinstance(segment, int) and isinstance(current, list):
            if segment < len(current):
                current = current[segment]
            else:
                break

    # Set the field to None
    last_segment = error.path[-1]
    if isinstance(last_segment, str):
        current[last_segment] = None
    elif isinstance(last_segment, int) and isinstance(current, list):
        if last_segment < len(current):
            current[last_segment] = None

    return result
```

D. Core Logic Skeleton Code

Parse Error Collection:

```python
# parser/parser.py

class Parser:

    def __init__(self, tokens: List[Token]):

        self.tokens = tokens

        self.position = 0

        self.errors: List[GraphQLSyntaxError] = []



    def parse_document(self) -> Document:

        """Parse complete GraphQL document with error collection."""

        # TODO 1: Initialize definitions list

        # TODO 2: While not at EOF, parse definition

        # TODO 3: If parse_definition returns None due to error,

        #         skip to next definition using error recovery

        # TODO 4: Implement error recovery: skip tokens until safe point

        #         (next '{', '}', or top-level keyword)

        # TODO 5: After parsing, if errors list not empty, raise first error

        #         or collect all for validation phase

        pass



    def expect(self, token_type: TokenType, value: Optional[str] = None) -> Token:

        """Expect specific token, record error if not found."""

        # TODO 1: Check if current token matches expected type/value

        # TODO 2: If match, consume token and return it

        # TODO 3: If no match, create GraphQLSyntaxError with:

        #         - Current token's location

        #         - Message: f"Expected {token_type}, found {actual_token.value}"

        #         - Add to self.errors list

        # TODO 4: Implement error recovery: return synthetic token

        pass
```

Execution Error Handling:

```python
# execution/executor.py

class Executor:

    def execute_field(self, parent_type: GraphQLType, field: Field,

                      parent_value: Any, path: List[PathSegment]) -> Any:

        """Execute single field with error handling."""

        # TODO 1: Get field definition from schema

        # TODO 2: Coerce arguments with try/except for coercion errors

        # TODO 3: Get resolver function

        # TODO 4: Try to execute resolver with try/except

        # TODO 5: If resolver raises exception:

        #         - Catch it and create GraphQLExecutionError

        #         - Add error to context.errors list

        #         - Check if field type is non-null using should_propagate_null

        #         - If non-null, return None to trigger propagation

        #         - If nullable, return None (field is null, error recorded)

        # TODO 6: If resolver returns value, validate it matches field type

        # TODO 7: Return resolved value

        pass
```

Database Error Sanitization:

```python
# errors/database_errors.py

def sanitize_database_error(error: Exception, sql: Optional[str] = None) -> GraphQLError:

    """Convert database exception to safe GraphQL error."""

    # TODO 1: Check error type (psycopg2.Error, sqlite3.Error, etc.)

    # TODO 2: Extract error message

    # TODO 3: Categorize based on patterns:

    #         - "duplicate key" -> constraint violation

    #         - "foreign key" -> relationship error

    #         - "syntax error" -> SQL compilation error

    #         - "connection" -> database unavailable

    # TODO 4: Create sanitized message per category

    # TODO 5: Include safe details in extensions (not in message)

    # TODO 6: Return GraphQLExecutionError

    pass
```

E. Python-Specific Hints

1. **Use context managers for database connections**:

```python
from contextlib import contextmanager


@contextmanager

def database_connection(config):

    conn = None

    try:

        conn = create_connection(config)

        yield conn

    except DatabaseError as e:

        raise sanitize_database_error(e)

    finally:

        if conn:

            conn.close()
```

2. **Leverage Python's exception chaining**:

```python
try:

    result = resolver(parent, args, context)

except Exception as e:

    raise GraphQLExecutionError(

        message=f"Resolver failed: {str(e)}",

        original_error=e  # Preserve original for logging

    ) from e
```

3. **Use `functools.lru_cache` for nullability checking**:

```python
from functools import lru_cache


@lru_cache(maxsize=128)

def is_non_null_type(graphql_type: GraphQLType) -> bool:

    """Memoized check for non-null types."""

    if isinstance(graphql_type, NonNullType):

        return True

    if isinstance(graphql_type, ListType):

        return is_non_null_type(graphql_type.of_type)

    return False
```

F. Milestone Checkpoint

After implementing error handling, verify with these test queries:

1. **Parse error checkpoint**:

```bash
python -m pytest tests/test_parser.py::test_syntax_errors -v
```

Expected: All 10 syntax error test cases pass with precise location information.

2. **Null propagation checkpoint**:

```bash
python -c "
from engine import execute_query, create_schema
schema = create_schema(...)
result = execute_query(schema, '{ user(id: 999) { name email } }')
print('Has errors:', bool(result.get('errors')))
print('User is null:', result.get('data', {}).get('user') is None)
"
```

Expected: When user 999 doesn't exist, `data.user` is `null` (not missing), and errors list contains appropriate error.

3. **Database error checkpoint**:

```bash
python -c "
from engine import reflect_schema
# Use invalid connection string
try:
    schema = reflect_schema('postgresql://invalid:invalid@localhost:9999/db')
except Exception as e:
    print('Error type:', type(e).__name__)
    print('Message contains password?', 'invalid' in str(e))
"
```

Expected: Error is raised, but error message doesn't contain password.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Query returns `{"data": null}` with no errors	Non-null field at root resolved to null	Check execution logs for which resolver returned null	Make resolver return value or change field to nullable
Error missing location information	Location not passed when creating error	Add `locations=[field.loc]` to error creation	Ensure AST nodes have location data
Multiple identical errors	Error propagating through list fields	Check if error is being added for each list element	Add error once at list level, not per element
Database error exposes table names	Insufficient error sanitization	Test with malformed query that causes SQL error	Implement error categorization in `sanitize_database_error`
Deep recursion crashes	No query depth limiting	Add depth counter to execution context	Implement `max_depth` validation
Cartesian product in results	Missing JOIN conditions in SQL compilation	Log generated SQL and examine JOIN clauses	Ensure every JOIN has ON condition

Debugging Guide

> **Milestone(s):** This section spans all five milestones, providing practical troubleshooting guidance for the most common issues learners encounter while building their GraphQL execution engine. Effective debugging requires understanding how errors manifest across the entire pipeline from query parsing to SQL execution.

Debugging a GraphQL engine involves tracing data through multiple transformation layers. When a query fails or returns unexpected results, the issue could originate in any component: a malformed AST from the parser, an inconsistent type definition, a resolver returning incorrect data, a mis-mapped database column, or an inefficient SQL query. This guide provides systematic approaches to isolate and fix the most common problems.

Common Bugs Table

When building a GraphQL engine, certain patterns of mistakes appear repeatedly across implementations. The following table categorizes symptoms by component, helping you quickly identify where to look based on what you're observing.

Symptom	Likely Cause	Diagnosis	Fix
Parser throws "Unexpected token" error on valid GraphQL	Tokenizer failing to recognize block strings, escaped strings, or unicode characters	1. Check tokenizer output for the query fragment around the error location 2. Verify string literal handling for escape sequences (`\n` , `\u0041`) 3. Test with simplified query removing complex string values	Implement proper string tokenization with full escape sequence support per GraphQL spec section 2.9
Query returns `null` for fields that should have data	1. Missing or misconfigured resolver function 2. Null propagation from non-null field error 3. Database column mismatch in SQL compilation	1. Check execution logs for resolver calls 2. Examine error list in `ExecutionResult` for hidden null propagation errors 3. Verify SQL query actually returns data for the field	1. Ensure resolvers are registered for all fields 2. Check non-null field resolvers don't return `None` 3. Verify column aliases in SQL match field names in result mapper
Nested relationship fields cause N+1 database queries	SQL compiler generating separate queries per parent instead of JOINs	1. Enable SQL logging and count queries for nested data 2. Check if `compile_to_sql` produces a single query with JOINs 3. Verify relationship detection found foreign keys	1. Implement proper join planning in `plan_joins` 2. Use `LATERAL JOIN` or batched IN queries for one-to-many relationships 3. Ensure `RelationshipAnalyzer` detects all foreign key constraints
Introspection query fails or returns incomplete schema	Missing or incorrect implementation of `__schema` , `__type` meta-fields	1. Test introspection query against a known working GraphQL server 2. Verify `Schema` type includes introspection types in `types` dictionary 3. Check that meta-field resolvers return properly formatted data	1. Implement `__Schema` , `__Type` , `__Field` , `__InputValue` types per GraphQL spec 2. Ensure introspection types are added to schema during `create_schema` 3. Verify meta-field resolvers handle all type kinds correctly
Mutation with variables receives "Variable '$var' is not defined"	Parser not extracting variable definitions from operation header	1. Inspect `OperationDefinition.variable_definitions` in parsed AST 2. Check variable definitions are being passed to execution context 3. Verify variable names match between definition and usage	1. Ensure `parse_operation_definition` extracts `VariableDefinition` nodes 2. Pass `variable_values` from request to `execute_query` 3. Validate variable usage against definitions before execution
Database reflection creates types with wrong scalar mappings	`TypeMapper.map_column_type` using incorrect type mappings for database dialect	1. Check raw `ColumnMetadata.data_type` from database introspection 2. Verify `POSTGRES_MAPPINGS` (or dialect equivalent) contains the type 3. Test with explicit type mappings in `ReflectionOptions`	1. Add missing type mappings to `TypeMapper` 2. Handle database-specific types (PostgreSQL `jsonb` , MySQL `TINYINT`) 3. Implement custom mapping registration for user-defined types
Fragment spreads cause "Unknown fragment" error	Fragment definitions not being collected into `fragments` dictionary	1. Check `Document.definitions` includes `FragmentDefinition` nodes 2. Verify fragment dictionary passed to `ExecutionContext` 3. Test with inline fragment (should work if named fragments fail)	1. Ensure `parse_document` collects fragments into separate dictionary 2. Pass fragments to execution context: `ExecutionContext.fragments` 3. Validate fragment spreads reference defined fragments before execution

Symptom	Likely Cause	Diagnosis	Fix
Query with nested lists returns incorrect nesting structure	Result mapper flattening nested structures incorrectly	1. Examine raw SQL result rows vs expected GraphQL response shape 2. Check `result_mapper` in `SQLQuery` properly reconstructs hierarchy 3. Verify join aliases preserve parent-child relationships	1. Implement result mapper that uses join aliases to reconstruct tree 2. Test with simple one-to-many relationship first 3. Use temporary table aliases that encode the selection path
Directives are ignored during execution	Directive parsing working but not applied during field resolution	1. Check `Directive` nodes appear in AST for fields/fragments 2. Verify directive information reaches resolver via `ExecutionContext` 3. Test with `@skip` or `@include` directives first	1. Implement directive evaluation before field resolution 2. Apply `@skip` / `@include` during selection set reduction 3. Pass directive arguments to resolver functions when needed
Non-null field error causes entire query to return `null`	Incorrect null propagation - null bubbling too far up	1. Check if error is in `ExecutionResult.errors` with path 2. Verify `should_propagate_null` correctly checks field type nullability 3. Test with nullable parent field (should return partial data)	1. Implement null propagation per GraphQL spec: only bubble to nearest nullable parent 2. Use `apply_null_propagation` to set nulls in result selectively 3. Ensure non-null errors are recorded but don't abort entire execution
SQL query vulnerable to injection via GraphQL arguments	Compiler using string concatenation instead of parameter binding	1. Check generated SQL for raw values in WHERE clause instead of `$1`, `?` 2. Verify `SQLParameter` list is populated with argument values 3. Test with malicious string argument containing SQL metacharacters	1. Use `SQLBuilder` with parameter placeholders for all user input 2. Ensure `compile_to_sql` returns both SQL string and parameters dictionary 3. Pass parameters separately to database driver's execute method
Self-referential relationship causes infinite recursion	Relationship detection creating circular reference without depth limit	1. Check `Relationship.is_self_referential` flag 2. Verify recursion depth limited in relationship traversal 3. Test with employee → manager → employee hierarchy	1. Implement cycle detection in `RelationshipAnalyzer` 2. Add `max_relationship_depth` to `ReflectionOptions` 3. Use iterative rather than recursive traversal for relationship chains
Interface/union types return "Abstract type must resolve to Object type"	Missing or incorrect `resolve_type` function	1. Check `InterfaceType.resolve_type` or `UnionType.resolve_type` is set 2. Verify resolver returns actual `ObjectType` not type name string 3. Test with concrete type query first, then through interface	1. Implement `resolve_type` that examines value and returns appropriate ObjectType 2. Ensure interface implementations properly registered in schema 3. Use `__typename` field in query to debug runtime type resolution
Pagination with cursors returns inconsistent results when data changes	Using OFFSET/LIMIT instead of cursor-based pagination	1. Check generated SQL for `OFFSET` clause 2. Verify cursor decoding/encoding uses stable row identifiers 3. Test with data insertion during pagination	1. Implement cursor pagination with unique, stable sort keys 2. Use `ORDER BY` with unique columns and WHERE clause for cursors 3. Encode cursor as base64 of sort column values

Debugging Techniques

Effective debugging requires more than just reading error messages—it involves instrumenting the system to observe internal state transitions. These techniques help you visualize what's happening at each stage of the GraphQL execution pipeline.

AST Visualization

> **Mental Model: Family Tree Diagram** - Think of the AST as a family tree where the query document is the ancestor, operations are the main branches, selection sets are sub-branches, and individual fields are leaves. Visualizing this tree helps you verify the parser correctly understood the query structure.

When the parser seems to be misinterpreting a query, the first step is to examine the AST it produced. A well-structured visualization reveals whether:

- Selection sets are properly nested
- Fragments are expanded correctly
- Directives are attached to the right nodes
- Variable definitions are captured

Diagnostic Approach:

1. **Print AST with Indentation:** Create a recursive function that prints each node type with increasing indentation for depth
2. **Highlight Key Attributes:** For each `Field` node, display `name`, `alias`, and argument count
3. **Track Locations:** Include `Location.line` and `Location.column` to correlate nodes with source text
4. **Compare with Reference:** Use a known working GraphQL implementation (like GraphQL.js) to parse the same query and compare structures

What to Look For:

- Are all expected fields present in the selection set?
- Do fragment spreads point to existing fragment definitions?
- Are argument values of the correct type (e.g., `IntValue` vs `StringValue`)?
- Do directives appear on the expected nodes with proper arguments?

SQL Logging

> **Mental Model: Kitchen Ticket System** - Imagine each GraphQL query generates a "ticket" that gets translated into cooking instructions (SQL). Logging the SQL shows you exactly what "cooking instructions" the kitchen is following, revealing whether the translation was correct.

SQL logging is critical for debugging the query compilation and N+1 problems. By intercepting and examining the actual SQL sent to the database, you can identify:

Implementation Strategy:

1. **Interceptor Pattern:** Wrap database connection execute methods to log SQL with timestamps
2. **Query Tagging:** Add a comment with GraphQL operation name to correlate SQL with GraphQL
3. **Parameter Substitution:** Log the actual parameter values (not placeholders) for complete queries
4. **Execution Metrics:** Capture query duration, row count, and explain plan results

Key Questions SQL Logging Answers:

- Is the compiler generating a single query with JOINs or multiple sequential queries?
- Are WHERE conditions correctly parameterized to prevent injection?
- Do pagination clauses (LIMIT/OFFSET) match GraphQL arguments?
- Are unnecessary columns being selected (SELECT * vs specific columns)?

Sample Log Format:

```
[GRAPHQL→SQL] Operation: GetUserWithPosts
SQL: SELECT u.id, u.name, p.id, p.title FROM users u
     LEFT JOIN posts p ON u.id = p.user_id WHERE u.id = $1
Parameters: [123]
Duration: 4.2ms, Rows: 5
```

Resolver Tracing

> **Mental Model: Function Call Stack Recording** - Imagine each resolver as a function call in a program. Tracing logs each call entry and exit with arguments and return values, creating a call graph that shows how data flows through resolvers.

Resolver tracing illuminates the execution phase, showing which resolvers are called, in what order, with what arguments, and what they return. This is particularly valuable for:

Tracing Implementation:

1. **Wrapper Functions:** Create a decorator/wrapper that logs resolver invocation
2. **Context Propagation:** Include unique request ID in `ExecutionContext` to correlate logs
3. **Timing Information:** Record start/end times to identify slow resolvers
4. **Error Capture:** Log exceptions before they're converted to GraphQL errors

Trace Data to Capture:

- Resolver function name or field path
- Parent value type and ID (if applicable)
- Argument values received
- Return value or exception
- Duration of resolution
- Whether resolution was batched via DataLoader

Diagnostic Patterns:

- **Missing Resolver Calls:** If a field returns null and its resolver never logged, the resolver wasn't registered
- **Wrong Argument Values:** If arguments in trace don't match expected types, check variable coercion
- **Multiple Calls for Same Data:** Indicates missing DataLoader batching
- **Long Resolver Chains:** Deeply nested synchronous resolvers causing performance issues

Type Inspection

> **Mental Model: Building Blueprint Examination** - Imagine the type system as architectural blueprints for a building. Type inspection lets you examine each blueprint's details: room dimensions (field types), connections (interfaces), and structural rules (nullability constraints).

When type-related errors occur ("Expected type String, found Int"), you need to examine the actual type definitions in the schema. Type inspection helps verify:

Inspection Tools:

1. **Schema Introspection Queries:** Use GraphQL's built-in introspection to query type definitions
2. **Programmatic Type Dumping:** Write a function that prints all types with their fields and relationships
3. **Validation Reports:** Run schema validation and examine all warnings/errors
4. **Comparison with Source:** Compare runtime type definitions with source schema definitions

Critical Type Details to Verify:

- Field nullability (`String` vs `String!` vs `[String]!`)
- Interface implementation completeness (all interface fields present)
- Union type membership (all possible types registered)
- Custom scalar serialization/parsing functions
- Default values for input fields and arguments

Common Type Inspection Findings:

- **Circular Input Types:** Input object fields that reference their containing type
- **Missing Interface Fields:** Object types claiming to implement interfaces but missing fields
- **Type Name Collisions:** Two types with same name but different definitions
- **Incorrect Type Modifiers:** List wrapped in NonNull instead of NonNull wrapped in List

Tools and Diagnostic Approaches

Beyond manual inspection, several systematic approaches and tools can accelerate debugging. These approaches provide different perspectives on the system's behavior.

Debug Logging with Context Correlation

> **Mental Model: Forensic Timeline Reconstruction** - Imagine each request as a crime scene. Debug logging creates a detailed timeline of events with evidence tags (request IDs) that let you reconstruct exactly what happened during each stage of processing.

Structured debug logging provides a comprehensive view of request processing across all components. The key is correlation: every log entry should include a request identifier that connects parser logs with executor logs with SQL logs.

Logging Strategy Components:

Log Level	Purpose	Example Output
DEBUG	Detailed internal state	`[DEBUG] Parser: Entering parse_selection_set at line 5, token: {`
INFO	Major component transitions	`[INFO] Executor: Starting execution of operation 'GetUser'`
WARN	Non-breaking issues	`[WARN] TypeMapper: Unknown database type 'jsonb', falling back to String`
ERROR	Failures requiring attention	`[ERROR] SQLCompiler: Cartesian product detected in join plan`

Context Correlation Implementation:

1. **Request ID Generation:** Create a unique ID at HTTP entry point
2. **Context Passing:** Include request ID in `ExecutionContext.context_value`
3. **Structured Logging:** Use JSON format with consistent fields: `timestamp`, `level`, `request_id`, `component`, `message`, `extra_data`
4. **Log Aggregation:** Collect logs from all components into a single stream

Diagnostic Queries You Can Answer:

- How long did each phase (parse, validate, execute, compile SQL) take for a specific request?
- Which component first encountered an error in a failing request?
- What was the exact SQL generated for a query that returned unexpected data?
- Which resolvers were called multiple times for the same data (N+1 pattern)?

GraphiQL Integration for Interactive Testing

> **Mental Model: Query Playground with X-Ray Vision** - GraphiQL is like a playground where you can try queries while wearing X-ray glasses that let you see not just the response, but also the execution path, timing, and internal state.

GraphiQL (or GraphQL Playground) provides an interactive environment to test queries with built-in diagnostics. Integrating your engine with GraphiQL gives you:

Enhanced GraphiQL Features:

1. **Query Validation:** Real-time schema validation and error highlighting
2. **Query History:** Compare current query with previous working queries
3. **Variable Input:** Test with different variable values without modifying query text
4. **Documentation Explorer:** Browse schema types and fields to verify introspection

Custom Extensions for Debugging:

1. **Execution Timing Extension:** Add query duration breakdown to response extensions
2. **SQL Preview Extension:** Show generated SQL alongside GraphQL query
3. **Resolver Trace Extension:** Include resolver call trace in response metadata
4. **AST Viewer Extension:** Display parsed AST tree structure

Integration Steps:

1. **Add GraphiQL HTML Endpoint:** Serve GraphiQL interface at `/graphiql` or `/playground`
2. **Implement Introspection:** Ensure `__schema` query works for documentation display
3. **Add Custom Extensions:** Extend `ExecutionResult.extensions` with debug information
4. **Enable Request Logging:** Log all GraphiQL queries for later analysis

Diagnostic Workflow with GraphiQL:

1. Start with a simple query that works in a known GraphQL server
2. Gradually add complexity (nested fields, fragments, variables)
3. Use query variables to test edge cases without modifying query structure
4. Examine response extensions for timing, SQL, and trace data
5. Compare with expected results from direct database queries

Database Explain Plans for Query Optimization

> **Mental Model: Car Engine Diagnostic Computer** - An EXPLAIN plan is like plugging a diagnostic computer into a car engine. It shows you exactly how the database plans to execute the query: which indexes it will use, join order, estimated costs, and potential bottlenecks.

When SQL queries are slow or inefficient, database EXPLAIN plans provide visibility into the database's execution strategy. This is particularly important for the SQL compiler component to verify it's generating optimal queries.

EXPLAIN Plan Analysis Workflow:

1. **Capture Generated SQL:** Use SQL logging to get the exact query with parameters
2. **Run EXPLAIN:** Execute `EXPLAIN (FORMAT JSON, ANALYZE) ...` on the query
3. **Interpret Results:** Look for:
   - **Seq Scans vs Index Scans:** Sequential table scans are expensive for large tables
   - **Join Types:** Nested loops vs hash joins vs merge joins
   - **Filter Estimates:** Whether row estimates match actual row counts
   - **Sort Operations:** In-memory sorts vs disk spills
   - **Total Cost:** Estimated execution cost in arbitrary units

Common SQL Compiler Issues Revealed by EXPLAIN:

- **Missing JOIN Conditions:** Causing Cartesian products (explode in row count)
- **Inefficient WHERE Clauses:** Not using indexes due to function wrapping or type mismatches
- **Unnecessary Columns:** Selecting columns not needed for response
- **Suboptimal Join Order:** Joining large tables before filtering them

Integration with GraphQL Engine:

1. **Explain Endpoint:** Add `/explain` endpoint that runs EXPLAIN on generated SQL
2. **Query Plan Caching:** Cache explain results for frequently generated query patterns
3. **Performance Regression Detection:** Compare explain plans between engine versions
4. **Index Recommendation:** Suggest database indexes based on frequent query patterns

Sample Diagnostic Process:

1. GraphQL query for users with their posts runs slowly
2. SQL logging shows query with 3 LEFT JOINs
3. EXPLAIN reveals nested loop join on large `posts` table

4. Diagnosis: Missing index on `posts.user_id`
5. Fix: Add index and verify new EXPLAIN plan uses index scan

Implementation Guidance

While the previous sections focused on conceptual debugging approaches, this section provides concrete implementation strategies for building the diagnostic tools themselves.

Technology Recommendations:

Diagnostic Need	Simple Option	Advanced Option
Logging	Python `logging` module with JSON formatter	Structured logging with `structlog` or `loguru`
AST Visualization	Recursive print functions with indentation	Graphviz DOT generation for visual graphs
SQL Interception	Database driver wrapper that logs before execute	SQL comment injection with query tagging
Request Tracing	Manual request ID generation and passing	OpenTelemetry integration with distributed tracing
Performance Profiling	`time.time()` manual instrumentation	`cProfile` for function-level profiling

Recommended File Structure for Diagnostics:

```
graphql-engine/
├── server.py                    # HTTP server with request ID middleware
├── engine/
│   ├── __init__.py
│   ├── parser.py                # Parser with debug logging
│   ├── types.py                 # Type system with validation
│   ├── executor.py              # Executor with resolver tracing
│   ├── compiler.py              # SQL compiler with explain capability
│   └── diagnostics/             # Diagnostic utilities
│       ├── __init__.py
│       ├── logger.py            # Structured logging setup
│       ├── ast_printer.py       # AST visualization
│       ├── sql_explainer.py     # EXPLAIN plan runner
│       ├── tracer.py            # Request tracing
│       └── profiler.py          # Performance profiling
└── examples/
    └── debug_queries.graphql    # Test queries for debugging
```

Diagnostic Logger Implementation:

PYTHON

```python
# engine/diagnostics/logger.py

import json

import logging

import uuid

from datetime import datetime

from typing import Any, Dict, Optional


class DiagnosticLogger:

    """Structured logger for GraphQL engine diagnostics."""



    def __init__(self, name: str, level: str = "INFO"):

        self.logger = logging.getLogger(name)

        self.logger.setLevel(getattr(logging, level))



        # JSON formatter for structured logs

        class JSONFormatter(logging.Formatter):

            def format(self, record: logging.LogRecord) -> str:

                log_entry = {

                    "timestamp": datetime.utcnow().isoformat() + "Z",

                    "level": record.levelname,

                    "logger": record.name,

                    "message": record.getMessage(),

                    "request_id": getattr(record, 'request_id', None),

                    "component": getattr(record, 'component', None),

                    "path": getattr(record, 'path', None),

                    **getattr(record, 'extra', {})

                }

                return json.dumps(log_entry)



        handler = logging.StreamHandler()

        handler.setFormatter(JSONFormatter())

        self.logger.addHandler(handler)



    def log(self, level: str, message: str,

            request_id: Optional[str] = None,

            component: Optional[str] = None,

            path: Optional[str] = None,
```

```
            **extra: Any) -> None:
        """Log with structured context."""
        extra_record = {
            'request_id': request_id,
            'component': component,
            'path': path,
            'extra': extra
        }
        self.logger.log(getattr(logging, level), message, extra=extra_record)

# Global diagnostic logger instance
diag_log = DiagnosticLogger("graphql-engine")

# Usage in components:
# diag_log.log("INFO", "Parsing query",
#              request_id=ctx.request_id,
#              component="parser",
#              query=query_str[:100])
```

AST Printer Implementation:

```python
# engine/diagnostics/ast_printer.py

from typing import Any, List

from ..parser import Document, Field, SelectionSet, Node


class ASTPrinter:

    """Prints AST in human-readable indented format."""



    def print_document(self, doc: Document, indent: int = 0) -> str:

        """TODO 1: Iterate through doc.definitions and print each"""

        # TODO 2: For OperationDefinition, print operation_type and name

        # TODO 3: For FragmentDefinition, print fragment name and type_condition

        # TODO 4: Call print_selection_set for each definition's selection_set

        # TODO 5: Include Location.line and Location.column for error correlation

        pass



    def print_selection_set(self, sel_set: SelectionSet, indent: int = 0) -> str:

        """TODO 1: Iterate through sel_set.selections"""

        # TODO 2: For Field nodes, print name, alias, and argument count

        # TODO 3: For FragmentSpread nodes, print ...name

        # TODO 4: For InlineFragment nodes, print ... on TypeCondition

        # TODO 5: Recursively print nested selection_sets with increased indent

        pass



    def print_field(self, field: Field, indent: int = 0) -> str:

        """TODO 1: Print field name with optional alias (alias: name)"""

        # TODO 2: Print arguments in parentheses if present

        # TODO 3: Print directives starting with @ if present

        # TODO 4: Recursively print selection_set if field has nested selections

        pass
```

SQL Explainer Implementation:

```python
# engine/diagnostics/sql_explainer.py

from typing import Dict, Any, Optional

from ..compiler import SQLQuery

import psycopg2  # or appropriate database driver


class SQLExplainer:

    """Runs EXPLAIN on generated SQL queries."""



    def __init__(self, db_connection):

        self.db_connection = db_connection



    def explain(self, sql_query: SQLQuery, analyze: bool = False) -> Dict[str, Any]:

        """TODO 1: Generate EXPLAIN query from SQLQuery"""

        # TODO 2: If analyze=True, add ANALYZE option (caution: executes query)

        # TODO 3: Execute EXPLAIN using database connection

        # TODO 4: Parse EXPLAIN output (JSON format if supported)

        # TODO 5: Extract key metrics: total cost, seq scans, index scans

        # TODO 6: Return structured explanation with warnings for inefficiencies

        pass



    def find_inefficiencies(self, plan: Dict[str, Any]) -> List[str]:

        """TODO 1: Check for sequential scans on large tables"""

        # TODO 2: Look for nested loop joins without indexes

        # TODO 3: Detect sorts that spill to disk

        # TODO 4: Identify underestimated row counts

        # TODO 5: Return list of specific inefficiency descriptions

        pass
```

Milestone Debugging Checkpoints:

After completing each milestone, run these specific tests to verify correctness:

Milestone 1 (Parser):

- Run: `python -m pytest tests/test_parser.py -v`
- Expected: All tests pass, including edge cases for strings with escapes
- Manual test: Parse complex query with fragments, variables, directives
- Diagnostic: Use `ASTPrinter` to visualize and compare with GraphQL.js output

Milestone 2 (Type System):

- Run: `python -m pytest tests/test_types.py -v`
- Expected: Schema validation passes for valid schemas, fails for invalid ones
- Manual test: Execute introspection query `{ __schema { types { name } } }`

- Diagnostic: Verify all built-in types appear in introspection

Milestone 3 (Execution):

- Run: `python -m pytest tests/test_executor.py -v`
- Expected: Queries return correct data, errors handled gracefully
- Manual test: Execute query with null propagation and partial results
- Diagnostic: Enable resolver tracing to verify call order and batching

Milestone 4 (Schema Reflection):

- Run: `python -m pytest tests/test_reflection.py -v`
- Expected: Generated schema matches database structure
- Manual test: Reflect a sample database with relationships
- Diagnostic: Compare reflected GraphQL types with database metadata

Milestone 5 (SQL Compilation):

- Run: `python -m pytest tests/test_compiler.py -v`
- Expected: Generated SQL is efficient and correct
- Manual test: Compile nested query and run EXPLAIN to verify joins
- Diagnostic: SQL logging shows single query with proper parameterization

Debugging Tips for Common Scenarios:

Symptom	Quick Diagnosis	Immediate Action
All queries return "Internal server error"	Check application logs for uncaught exceptions	Add try-except around `execute_query` and log exception details
Introspection works but queries fail	Likely resolver not registered for fields	Verify field resolvers attached during schema construction
Query slow with deep nesting	Enable SQL logging, check for N+1 queries	Implement DataLoader batching for relationship fields
Mutation arguments not received	Check variable definitions parsing	Debug `parse_operation_definition` for variable extraction
Database type maps incorrectly	Examine `ColumnMetadata.data_type` raw value	Add custom mapping to `TypeMapper.register_custom_mapping`

Remember that debugging is iterative: make one change at a time, test thoroughly, and use the diagnostic tools to verify the fix actually addresses the root cause. The layered architecture means issues can cascade, so always trace problems back to their origin component.

Future Extensions

> **Milestone(s):** This section spans all five milestones, looking beyond the core implementation to consider how the architecture accommodates evolution. While these features are explicitly out of scope for the initial implementation, the system has been designed with clear extension points for future development.

The current GraphQL engine provides a solid foundation for executing queries against relational databases, but its architecture deliberately leaves room for expansion. Designing for extensibility from the beginning ensures that future features can be added with minimal disruption to the core system. This section explores four high-value extension points—schema stitching, subscriptions, custom directives, and query caching—and explains how the current architecture accommodates them through careful abstraction boundaries, pluggable components, and thoughtful data model design.

Planned Extension Points

Four major extension areas represent natural evolutions of the GraphQL engine that align with real-world GraphQL deployment patterns. Each builds upon the existing architecture while introducing new capabilities.

Schema Stitching: Federating Multiple Data Sources

Mental Model: Mosaic Artist Think of schema stitching as a mosaic artist who creates a unified picture from distinct pieces of colored glass. Each piece (individual GraphQL schema) has its own shape and color, but when arranged together with careful planning, they form a cohesive whole that appears as a single image to the viewer.

Schema stitching allows the GraphQL engine to combine multiple GraphQL schemas—potentially from different databases, REST APIs, or other GraphQL services—into a single unified schema. This transforms the engine from a single-database query layer into a federated gateway that can aggregate data from multiple sources.

Extension Requirements:

- Schema composition mechanism that merges type definitions and resolves naming conflicts
- Query planning that routes sub-queries to appropriate underlying services
- Field resolver delegation that forwards requests to remote services with appropriate arguments
- Error aggregation from multiple sources with proper source attribution

Implementation Approach: The system would extend the current `Schema` type with composition metadata and introduce a `SchemaComposer` component that:

1. Validates schema compatibility (no type conflicts, consistent interface implementations)
2. Creates merged type definitions with source service annotations
3. Generates delegation resolvers that forward field requests to appropriate services
4. Implements query planning to minimize remote calls and batch where possible

Architecture Decision Record: Schema Composition Strategy

> **Decision: Preserve Individual Schemas with Gateway Layer**
>
> - **Context**: When combining multiple GraphQL schemas, we need to decide whether to merge them at the type definition level or keep them separate with a routing layer.
> - **Options Considered**:
>   1. **Deep Merge**: Combine all type definitions into a single monolithic schema, resolving conflicts through renaming or namespacing.
>   2. **Gateway with Delegation**: Maintain separate schemas with a gateway layer that routes field resolutions to appropriate services.
>   3. **Schema Extensions**: Use GraphQL schema extensions to add cross-service fields while keeping core schemas independent.
> - **Decision**: Gateway with delegation, implemented as a separate `SchemaStitchingGateway` component.
> - **Rationale**: This approach preserves service autonomy, allows independent schema evolution, and matches the microservices architecture pattern where each service owns its domain. The delegation model also enables progressive migration from single-database to federated queries.
> - **Consequences**: Requires additional query planning complexity, introduces network latency for cross-service calls, but provides clear service boundaries and incremental adoption path.

Option	Pros	Cons	Selected?
Deep Merge	Simple query execution, single schema validation	Breaks service autonomy, complex conflict resolution	No
Gateway with Delegation	Service independence, incremental adoption	Query planning complexity, network overhead	**Yes**
Schema Extensions	GraphQL-spec compliant, clean separation	Limited tooling support, complex implementation	No

Subscriptions: Real-Time Data Updates

Mental Model: Radio Broadcasting Think of subscriptions as a radio broadcasting system where listeners tune into specific channels. The broadcaster (GraphQL engine) transmits updates only to those subscribed to particular data changes, without requiring continuous polling from listeners. Each subscription establishes a persistent connection over which data updates flow as they occur.

Subscriptions extend the GraphQL engine beyond request-response to support real-time data updates through WebSocket or Server-Sent Events connections. This enables applications to receive live updates when database data changes, such as new chat messages, stock price changes, or collaborative editing events.

Extension Requirements:

- WebSocket or SSE transport layer for persistent bidirectional connections
- Subscription operation parsing and validation in the parser
- Pub/sub mechanism for broadcasting data changes to interested subscribers
- Database change detection through triggers, polling, or database notifications
- Subscription execution context distinct from query/mutation contexts

Implementation Approach: The system would extend the current architecture with:

1. **Transport Layer**: WebSocket server that manages connection lifecycle and maps messages to subscription operations
2. **Subscription Resolver Type**: New resolver signature that returns an async iterator/generator instead of a value
3. **Event System**: Pub/sub system where database change events trigger subscription updates
4. **Subscription Manager**: Tracks active subscriptions, manages cleanup, and handles fan-out to multiple clients

Subscription Component	Integration Point	Data Flow
WebSocketTransport	Extends HTTP server layer	Handles connection upgrade, message framing, and protocol negotiation
SubscriptionParser	Extends GraphQL Parser (Component 1)	Adds subscription operation type validation and subscription-specific syntax
SubscriptionExecutor	Extends Query Execution Engine (Component 3)	Executes subscription resolvers, returns async iterators, manages event mapping
DatabaseChangeDetector	Integrates with Database Schema Reflection (Component 4)	Listens to database changes via triggers, LISTEN/NOTIFY, or change data capture
SubscriptionRegistry	New standalone component	Maps database changes to active subscriptions, manages fan-out and cleanup

Custom Directives: Schema-Level Behavior Modification

Mental Model: Circuit Breaker Panel Think of custom directives as circuit breakers in an electrical panel that can be flipped to modify how power flows through specific circuits. Each directive intercepts the normal execution flow at a specific point (field resolution, argument coercion, etc.) and can modify behavior—adding authentication checks, transforming output, logging, or implementing rate limiting—without changing the underlying resolver logic.

Custom directives extend the GraphQL engine's ability to modify execution behavior at the schema level through declarative annotations. This enables cross-cutting concerns like authentication, authorization, caching, logging, and data transformation to be expressed directly in the schema definition language.

Extension Requirements:

- Directive definition and validation in the type system
- Directive execution hooks at various points in the query lifecycle
- Directive composition and ordering semantics
- Directive argument processing and validation

Implementation Approach: The system would extend the current architecture with:

1. **Directive Definition Language**: Extend `GraphQLDirective` type to support execution hooks
2. **Directive Execution Pipeline**: Interceptors at parse, validate, and execute phases
3. **Directive Registry**: Central registry of available directives with their implementation
4. **Schema Transformation**: Apply directive semantics during schema building

Directive Execution Hook Points:

Hook Point	Description	Use Cases
`onSchema`	Applied to entire schema	Schema-level metadata, global rate limiting
`onObjectType`	Applied to object type definitions	Type-level permissions, data masking rules
`onFieldDefinition`	Applied to field definitions	Field-level authorization, cost calculation
`onArgumentDefinition`	Applied to argument definitions	Input validation, sanitization
`onFieldExecution`	Wraps field resolver execution	Caching, logging, timing, error handling
`onQueryExecution`	Wraps entire query execution	Request logging, tracing, overall timeout

Query Caching: Performance Optimization Layer

Mental Model: Library Index System Think of query caching as a library's card catalog system that remembers where to find frequently requested books. Instead of walking through the entire library (executing the full query) each time someone asks for popular books, the system maintains a quick-reference index (cache) that dramatically reduces lookup time for repeated requests with identical parameters.

Query caching adds a performance optimization layer that stores and reuses query results to reduce database load and improve response times. This includes both query result caching (storing complete GraphQL responses) and field-level caching (storing individual field values for reuse across queries).

Extension Requirements:

- Cache key generation from query AST, variables, and context
- Cache storage backend abstraction (in-memory, Redis, memcached)
- Cache invalidation strategies based on database mutations
- Cache TTL management and stale-while-revalidate patterns
- Partial query caching and result composition

Implementation Approach: The system would extend the current architecture with:

1. **Cache Key Generator**: Creates deterministic cache keys from queries
2. **Cache Layer Interface**: Abstract interface supporting multiple storage backends
3. **Cache Interceptor**: Wraps execution engine to check cache before execution
4. **Invalidation System**: Listens to mutations and invalidates affected cache entries

Cache Strategy Comparison:

Strategy	Implementation Complexity	Cache Hit Rate	Invalidation Complexity	Best For
Full Query Caching	Low	Medium	High	Read-heavy workloads with limited query variation
Field-Level Caching	High	High	Medium	Complex queries with overlapping field selections
Compiled SQL Caching	Medium	High	Low	Database-heavy workloads with repeated query patterns
Hybrid Approach	Very High	Very High	High	Production systems with mixed access patterns

> **Design Insight**: The choice between these caching strategies depends heavily on the access pattern. For GraphQL APIs with high query diversity, field-level caching often provides the best balance, while for APIs with repetitive query patterns (like dashboard applications), full query caching may be more effective.

Design Accommodations

The current architecture has been intentionally designed with specific patterns and abstractions that enable these future extensions without requiring fundamental redesign. These accommodations manifest as extension points, pluggable interfaces, and carefully isolated concerns.

Extension Point Inventory

The following table catalogs the explicit extension points built into the current system design:

Extension Point	Location	Purpose	Enables Feature
`GraphQLDirective.execution_hooks`	`GraphQLDirective` type	Directive implementation hooks	Custom directives
`Schema.builders`	`Schema` type	Schema transformation pipeline	Schema stitching, custom directives
`Executor.middleware`	`ExecutionContext`	Execution interception	Custom directives, caching, logging
`SQLCompiler.plugins`	`SQLBuilder`	SQL generation customization	Database-specific optimizations
`TypeMapper.registry`	`TypeMapper` class	Custom type mappings	Extended database type support
`Transport.protocols`	HTTP server layer	Alternative transport protocols	Subscriptions (WebSocket/SSE)
`DataLoader.factory`	`ExecutionContext`	Custom batching strategies	Advanced N+1 optimization
`ReflectionOptions.hooks`	`ReflectionOptions`	Schema generation customization	Complex relationship modeling

Abstract Interface Design

Each major component exposes abstract interfaces that allow behavior extension through polymorphism rather than modification:

1. Transport Layer Abstraction:

```python
# Current implementation supports only HTTP POST

# Extended to support WebSocket and Server-Sent Events

class TransportProtocol(ABC):

    @abstractmethod

    async def handle_request(self, request) -> Response:

        pass



    @abstractmethod

    def supports_subscriptions(self) -> bool:

        return False
```

2. Cache Backend Abstraction:

```python
# Prepared for caching layer addition
class CacheBackend(ABC):
    @abstractmethod
    async def get(self, key: str) -> Optional[Any]:
        pass

    @abstractmethod
    async def set(self, key: str, value: Any, ttl: Optional[int] = None):
        pass

    @abstractmethod
    async def invalidate_pattern(self, pattern: str):
        pass
```

3. Directive Execution Abstraction:

```python
# Ready for directive system extension
class DirectiveVisitor(ABC):
    @abstractmethod
    def visit_field_definition(self, field: GraphQLField, directive: GraphQLDirective):
        pass

    @abstractmethod
    def visit_argument_definition(self, arg: GraphQLArgument, directive: GraphQLDirective):
        pass
```

Schema Evolution Support

The type system design supports schema evolution through several key mechanisms:

1. Schema Versioning Metadata: The `Schema` type includes extension fields that can store version information and compatibility metadata:

Field	Type	Description	Evolution Support
`extensions`	`Dict[str, Any]`	Arbitrary metadata	Stores schema version, deprecation info
`source_services`	`List[Dict]`	Source service info	Enables schema stitching tracking
`directive_locations`	`Dict[str, List[str]]`	Custom directive bindings	Supports directive registration

2. Backward-Compatible Changes: The architecture supports common GraphQL schema evolution patterns:

- **Additive Changes**: New types, fields, and arguments can be added without breaking existing clients
- **Deprecation Flow**: Fields can be marked deprecated while maintaining functionality
- **Default Values**: New arguments can be added with default values for backward compatibility
- **Interface Implementation**: New types can implement existing interfaces

3. Breaking Change Detection: The `validate_schema` function includes hooks for schema comparison that can detect breaking changes:

```python
# Extension point for schema comparison
def detect_breaking_changes(
    old_schema: Schema,
    new_schema: Schema
) -> List[BreakingChange]:
    # Compares schemas and returns list of breaking changes
    # Ready for extension to support custom breaking change policies
    pass
```

Execution Pipeline Extensibility

The query execution pipeline is designed as a series of interceptable stages:

Execution Stage Pipeline:

1. **Parse** → `parse_query()` with directive awareness
2. **Validate** → `validate_with_rules()` with custom validation rules
3. **Plan** → `create_execution_plan()` with caching/sharding decisions
4. **Execute** → `execute_operation()` with middleware wrapping
5. **Format** → `format_response()` with error transformation

Each stage can be intercepted or replaced through the `ExecutionContext` configuration:

```python
# Execution context includes extension points
ExecutionContext(
    # ... existing fields ...
    validation_rules: List[ValidationRule] = None,  # Custom validation
    field_middleware: List[FieldMiddleware] = None,  # Field execution hooks
    query_transformers: List[QueryTransformer] = None,  # AST transformations
    error_formatters: Dict[Type[Exception], ErrorFormatter] = None,  # Custom errors
)
```

Database Abstraction Layer

The current database reflection and SQL compilation components are designed with dialect abstraction that enables future database support:

Database Dialect Interface:

```python
class SQLDialect(ABC):

    @abstractmethod

    def get_type_mappings(self) -> Dict[str, GraphQLType]:

        """Return database-specific type mappings"""

        pass



    @abstractmethod

    def supports_feature(self, feature: str) -> bool:

        """Check if dialect supports specific feature (e.g., JSON, arrays)"""

        pass



    @abstractmethod

    def generate_pagination_sql(self, limit: int, offset: int) -> str:

        """Generate dialect-specific pagination SQL"""

        pass
```

Current dialect implementations can be extended for:

- **Additional Databases**: MongoDB, Cassandra, Elasticsearch
- **Advanced Features**: Full-text search, geospatial queries, window functions
- **Optimization Hints**: Database-specific query optimizations

Plugin System Architecture

The overall architecture follows a plugin system pattern where new capabilities can be registered without modifying core components:

Plugin Registration Flow:

1. **Discovery**: Plugins are discovered via entry points or explicit registration
2. **Initialization**: Each plugin receives configuration and context
3. **Integration**: Plugins register themselves with appropriate extension points
4. **Execution**: Plugins participate in the normal execution flow

Example Plugin Contract:

```python
class GraphQLEnginePlugin(ABC):

    @abstractmethod

    def configure_schema(self, schema: Schema) -> Schema:

        """Transform schema during construction"""

        pass



    @abstractmethod

    def configure_execution(self, context: ExecutionContext) -> ExecutionContext:

        """Add middleware or transformers to execution context"""

        pass



    @abstractmethod

    def teardown(self):

        """Clean up plugin resources"""

        pass
```

Performance Monitoring Hooks

The architecture includes performance monitoring hooks that enable future observability features:

Instrumentation Points:

Instrumentation Point	Data Collected	Enables Feature
`query_parse_start/end`	Parse duration, query size	Performance monitoring
`field_resolve_start/end`	Resolver timing, parent type	Resolver profiling
`sql_generation_start/end`	SQL generation time, complexity	Query optimization
`database_query_start/end`	Database latency, row count	Database performance analysis
`cache_hit_miss`	Cache effectiveness metrics	Cache tuning

These hooks are implemented as no-op stubs in the current system but provide the foundation for comprehensive APM (Application Performance Monitoring) integration.

Security Extension Foundation

The current design includes several security-conscious patterns that enable future security extensions:

1. Input Validation Pipeline: Argument coercion and input validation occur in isolated, interceptable stages that can be extended for:

- Additional input sanitization
- Business rule validation
- Compliance checking (PCI, HIPAA, etc.)

2. Error Sanitization Layers: Database errors are sanitized before exposure, but this pipeline can be extended for:

- PII (Personally Identifiable Information) filtering
- Security policy enforcement
- Audit logging of sensitive errors

3. Query Complexity Analysis: The query execution plan includes complexity metrics that can be extended for:

- Query depth limiting
- Field cost calculation
- Rate limiting based on query complexity

Common Pitfalls in Extensibility Design: ⚠️ Pitfall: Over-Engineering Extension Points

- **Description**: Creating abstract interfaces and extension hooks for every possible future need, resulting in unnecessary complexity.
- **Why It's Wrong**: Increases initial implementation complexity, makes the system harder to understand, and may never use most extension points.
- **Fix**: Apply the YAGNI (You Ain't Gonna Need It) principle. Only add extension points where there's clear evidence of future need or where the cost is minimal.

⚠️ Pitfall: Tight Coupling Between Extension and Core

- **Description**: Extension implementations directly modify core data structures or rely on internal implementation details.
- **Why It's Wrong**: Core changes break extensions, making evolution difficult and creating maintenance burden.
- **Fix**: Use the Dependency Inversion Principle. Core depends on abstractions, extensions implement those abstractions without knowing core internals.

⚠️ Pitfall: Inconsistent Extension Patterns

- **Description**: Different components use different patterns for extensibility (callbacks, inheritance, plugins, etc.).
- **Why It's Wrong**: Makes the system harder to learn and extend consistently.
- **Fix**: Establish and document consistent extension patterns across the codebase, such as a unified plugin system or consistent use of strategy pattern.

⚠️ Pitfall: Neglecting Extension Performance

- **Description**: Extension points add overhead even when not used, or poorly designed extensions degrade performance.
- **Why It's Wrong**: Impacts performance of core functionality, making extensions impractical for production use.
- **Fix**: Design extension points with zero-cost abstractions where possible, use lazy initialization, and provide performance guidance for extension developers.

> **Design Insight**: The most successful extensible systems follow the "Open-Closed Principle"—open for extension but closed for modification. This is achieved not by predicting every future need, but by identifying stable abstractions that encapsulate variation points. In this GraphQL engine, the key stable abstractions are: `GraphQLType` for schema definition, `ExecutionContext` for execution behavior, and `SQLDialect` for database variations.

Implementation Guidance

While the future extensions themselves are out of scope for the initial implementation, the architecture includes specific patterns and structures that make these extensions feasible. This section provides concrete guidance on how to implement the extension infrastructure.

Technology Recommendations

Extension	Simple Option	Advanced Option
Schema Stitching	Manual schema merging with delegated resolvers	Apollo Federation with query planning service
Subscriptions	Polling-based subscriptions with database polling	WebSocket with PostgreSQL LISTEN/NOTIFY or change data capture
Custom Directives	Directive visitors that modify execution context	AST-transforming directives with full query rewriting
Query Caching	In-memory LRU cache with TTL	Redis cluster with cache invalidation streams
Plugin System	Explicit registration in application code	Dynamic discovery via entry_points with dependency injection

Recommended Plugin Directory Structure

```
graphql-engine/
  core/                    # Core engine (existing)
    parser/               # Component 1
    type_system/          # Component 2
    execution/            # Component 3
    reflection/           # Component 4
    compilation/          # Component 5

  extensions/             # Future extensions (new)
    stitching/            # Schema stitching implementation
      __init__.py
      gateway.py          # Schema stitching gateway
      delegator.py        # Field delegation logic
      planner.py          # Cross-service query planning

    subscriptions/        # Real-time subscriptions
      __init__.py
      transport/          # WebSocket/SSE transport
        websocket.py
        sse.py
      pubsub/             # Publish/subscribe backend
        memory.py
        redis.py
      triggers/           # Database change detection
        poller.py
        cdc.py

    directives/           # Custom directives
      __init__.py
      registry.py         # Directive registry
      builtins/           # Built-in directives
        auth.py           # @auth directive
        cache.py          # @cache directive
        log.py            # @log directive
      middleware.py       # Directive execution middleware

    caching/              # Query caching
      __init__.py
      backends/           # Cache storage backends
        memory.py
        redis.py
      key_generation.py   # Cache key generation
      invalidation.py     # Cache invalidation logic

    plugins/              # Plugin infrastructure
      __init__.py
      registry.py         # Plugin registry
      lifecycle.py        # Plugin lifecycle management
```

Extension Infrastructure Starter Code

Plugin Registry (Complete Implementation):

PYTHON

```python
# extensions/plugins/registry.py
"""Plugin registry for dynamic extension loading"""
import importlib
import logging
from typing import Dict, List, Type, Optional, Any

logger = logging.getLogger(__name__)

class GraphQLEnginePlugin:
    """Base class for all GraphQL engine plugins"""

    def __init__(self, name: str, version: str = "1.0.0"):
        self.name = name
        self.version = version
        self.enabled = True

    def configure_schema(self, schema: 'Schema') -> 'Schema':
        """Transform schema during construction. Override in subclasses."""
        return schema

    def configure_execution(self, context: 'ExecutionContext') -> 'ExecutionContext':
        """Add middleware or transformers to execution context."""
        return context

    def teardown(self):
        """Clean up plugin resources."""
        pass

    def __repr__(self) -> str:
        return f"<Plugin {self.name} v{self.version}>"

class PluginRegistry:
    """Central registry for managing plugins"""

    def __init__(self):
        self._plugins: Dict[str, GraphQLEnginePlugin] = {}
        self._initialized = False
```

```
def register(self, plugin: GraphQLEnginePlugin) -> None:
    """Register a plugin instance"""
    if plugin.name in self._plugins:
        logger.warning(f"Plugin {plugin.name} already registered, overwriting")
    self._plugins[plugin.name] = plugin

def load_from_entrypoint(self, entrypoint_name: str = "graphql_engine.plugins") -> None:
    """Load plugins from package entrypoints"""
    try:
        import pkg_resources
        for entrypoint in pkg_resources.iter_entry_points(entrypoint_name):
            try:
                plugin_class = entrypoint.load()
                plugin = plugin_class()
                self.register(plugin)
                logger.info(f"Loaded plugin {plugin.name} from {entrypoint.module_name}")
            except Exception as e:
                logger.error(f"Failed to load plugin from {entrypoint}: {e}")
    except ImportError:
        logger.warning("pkg_resources not available, skipping entrypoint loading")

def initialize_all(self, schema: 'Schema', context: Optional['ExecutionContext'] = None) -> None:
    """Initialize all registered plugins"""
    if self._initialized:
        return

    for name, plugin in self._plugins.items():
        if plugin.enabled:
            try:
                logger.debug(f"Initializing plugin {name}")
                # Apply schema transformations
                schema = plugin.configure_schema(schema)

                # Apply execution context transformations
                if context:
```

```
                    context = plugin.configure_execution(context)

            except Exception as e:
                logger.error(f"Plugin {name} failed to initialize: {e}")
                plugin.enabled = False

    self._initialized = True

def teardown_all(self) -> None:
    """Clean up all plugin resources"""
    for name, plugin in self._plugins.items():
        if plugin.enabled:
            try:
                plugin.teardown()
            except Exception as e:
                logger.error(f"Plugin {name} failed to teardown: {e}")

def get_plugin(self, name: str) -> Optional[GraphQLEnginePlugin]:
    """Get a plugin by name"""
    return self._plugins.get(name)

def list_plugins(self) -> List[GraphQLEnginePlugin]:
    """List all registered plugins"""
    return list(self._plugins.values())

def enable_plugin(self, name: str) -> bool:
    """Enable a disabled plugin"""
    if name in self._plugins:
        self._plugins[name].enabled = True
        return True
    return False

def disable_plugin(self, name: str) -> bool:
    """Disable an enabled plugin"""
    if name in self._plugins:
        self._plugins[name].enabled = False
```

```
            return True

        return False


# Global plugin registry instance

plugin_registry = PluginRegistry()
```

Directive Middleware Skeleton (Core Logic TODOs):

PYTHON

```python
# extensions/directives/middleware.py

"""Directive execution middleware for field resolution"""

from typing import Callable, Any, Dict, List, Optional

from ..core.type_system import GraphQLDirective, GraphQLField, ExecutionContext


class DirectiveMiddleware:

    """Middleware that executes directives during field resolution"""


    def __init__(self):

        self._directive_handlers: Dict[str, Callable] = {}


    def register_directive_handler(self, directive_name: str, handler: Callable) -> None:

        """Register a handler function for a specific directive"""

        # TODO 1: Validate handler signature matches expected directive interface

        # TODO 2: Check for naming conflicts with existing handlers

        # TODO 3: Store handler in _directive_handlers dictionary

        pass


    async def execute_field_with_directives(

        self,

        field: GraphQLField,

        parent_value: Any,

        args: Dict[str, Any],

        context: ExecutionContext,

        next_resolver: Callable

    ) -> Any:

        """Execute a field with directive middleware wrapping"""

        # TODO 1: Collect all directives attached to this field definition

        # TODO 2: Sort directives by their declared execution order

        # TODO 3: Create execution chain: directive1 → directive2 → ... → resolver

        # TODO 4: Execute the chain and return result

        # TODO 5: Handle errors from directive execution with proper error propagation

        pass


    def _create_directive_wrapper(

        self,

        directive: GraphQLDirective,
```

```
        handler: Callable,
        next_fn: Callable
    ) -> Callable:
        """Create a wrapper function that executes a directive"""
        # TODO 1: Extract directive arguments from directive definition
        # TODO 2: Create closure that captures directive context
        # TODO 3: Return function that:
        #   - Calls handler with appropriate parameters
        #   - Passes result to next_fn in chain
        #   - Handles exceptions according to directive configuration
        pass
```

Subscription Transport Skeleton (Core Logic TODOs):

PYTHON

```python
# extensions/subscriptions/transport/websocket.py

"""WebSocket transport for GraphQL subscriptions"""

import asyncio

import json

from typing import Dict, Any, Optional

from websockets.server import WebSocketServerProtocol


class WebSocketTransport:

    """WebSocket-based transport for GraphQL subscriptions"""



    def __init__(self, execution_context_factory: Callable):

        # TODO 1: Store execution context factory for creating per-connection contexts

        # TODO 2: Initialize connection tracking dictionary

        # TODO 3: Set up message handlers for different GraphQL over WebSocket protocols

        pass



    async def handle_connection(self, websocket: WebSocketServerProtocol) -> None:

        """Handle a new WebSocket connection"""

        # TODO 1: Generate unique connection ID

        # TODO 2: Store connection in active connections registry

        # TODO 3: Send connection_init acknowledgement

        # TODO 4: Enter message handling loop

        # TODO 5: On connection_close message or disconnect, clean up resources

        pass



    async def handle_message(

        self,

        connection_id: str,

        message: Dict[str, Any]

    ) -> Optional[Dict[str, Any]]:

        """Handle incoming WebSocket message"""

        # TODO 1: Parse message type (connection_init, start, stop, connection_terminate)

        # TODO 2: For "start" messages: parse GraphQL operation, validate for subscriptions

        # TODO 3: Create subscription execution, store subscription ID

        # TODO 4: Return appropriate response based on protocol specification

        # TODO 5: Handle errors with proper error response format

        pass
```

```
    async def send_data(
        self,
        connection_id: str,
        subscription_id: str,
        data: Dict[str, Any]
    ) -> None:
        """Send subscription data to client"""
        # TODO 1: Look up WebSocket connection by connection_id
        # TODO 2: Format data according to GraphQL over WebSocket specification
        # TODO 3: Send data message with subscription_id and payload
        # TODO 4: Handle send errors (connection closed, etc.)
        pass
```

Language-Specific Hints for Python

1. **Use `__init_subclass__` for Plugin Registration**: Python 3.6+ supports `__init_subclass__` which can automatically register plugin classes without explicit registration calls.
2. **Leverage `contextvars` for Request-Scoped Data**: For directive middleware that needs request-scoped context, use `contextvars` instead of thread locals for proper async support.
3. **Use `dataclasses` for Configuration Objects**: Future extension configurations should use `@dataclass` for type safety and clean initialization.
4. **Implement `__slots__` for Performance-Critical Types**: Extension data structures that are instantiated frequently should use `__slots__` to reduce memory overhead.
5. **Use `asyncio.Queue` for Subscription Events**: For subscription implementations, `asyncio.Queue` provides a clean pattern for buffering events between database change detection and client delivery.

Extension Milestone Checkpoint

After implementing the extension infrastructure (not the full extensions), verify the system still functions correctly:

1. **Run Existing Tests**: Execute the full test suite to ensure no regression:

```
python -m pytest tests/ -v
```

2. **Verify Plugin Loading**: Test that plugins can be registered and initialized:

```python
# test_plugin_loading.py

from extensions.plugins.registry import plugin_registry, GraphQLEnginePlugin


class TestPlugin(GraphQLEnginePlugin):

    def __init__(self):

        super().__init__("test-plugin")


    def configure_schema(self, schema):

        print(f"TestPlugin configuring schema: {schema}")

        return schema


# Register and test

plugin_registry.register(TestPlugin())

print(f"Registered plugins: {plugin_registry.list_plugins()}")
```

3. **Check Directive Hook Integration**: Verify directives can be attached to schema types:

```python
# test_directive_hooks.py

schema = create_schema(...)

# Should be able to attach directives without errors
```

4. **Monitor Performance Impact**: Run performance benchmarks to ensure extension infrastructure doesn't degrade performance:

```
python benchmarks/query_performance.py --baseline
```

Expected Behavior:

- All existing tests pass without modification
- Plugin registration and initialization logs appropriate messages
- Schema creation with directive annotations succeeds
- Performance impact is <5% for non-extension use cases

Signs of Problems:

- Existing tests fail after adding extension infrastructure → Likely caused by side effects in plugin initialization
- Schema building fails with directive annotations → Directive integration not properly isolated
- Significant performance degradation → Extension hooks adding overhead in hot paths

Debugging Extension Implementation

Symptom	Likely Cause	How to Diagnose	Fix
Plugins not loading	Entrypoint name mismatch or missing dependency	Check plugin registry logs, verify `setup.py` entrypoints	Correct entrypoint name, ensure package is installed
Directive not executing	Directive not registered or middleware not connected	Add debug logging to directive middleware, check execution context	Register directive handler, ensure middleware is in execution chain
Subscription connections dropping	WebSocket protocol violation or timeout	Monitor WebSocket frames, check for keepalive messages	Implement proper ping/pong, adjust timeout settings
Cache returning stale data	Cache invalidation not triggered by mutations	Log cache operations, check invalidation triggers	Connect mutation execution to cache invalidation system
Schema stitching type conflicts	Type merging logic too restrictive	Log type merging decisions, examine conflict details	Implement type namespacing or field renaming strategies
Performance degradation with extensions	Extension hooks in hot paths	Profile execution with and without extensions, identify bottlenecks	Optimize extension hooks, add conditional execution
Memory leak with plugins	Plugin resources not cleaned up	Monitor memory usage, check plugin teardown methods	Ensure all plugins implement proper teardown, use weak references

> **Key Insight**: The most successful extension systems evolve from actual needs rather than hypothetical ones. Implement the minimal extension infrastructure needed for known requirements, then extend as real use cases emerge. The architecture should make common extensions easy and complex extensions possible, without making the simple case complex.

Glossary

> **Milestone(s):** This glossary section spans all five milestones, providing definitions for key terms used throughout the design document. Understanding this vocabulary is essential for navigating the technical concepts, data structures, and implementation patterns across all components.

This glossary defines the specialized terminology, acronyms, and domain vocabulary used throughout the GraphQL engine design document. Terms are organized by conceptual domain (GraphQL, database, compiler) to help you quickly locate definitions when encountering unfamiliar concepts in the implementation guidance or design discussions.

GraphQL Terminology

GraphQL-specific terms that describe the query language, type system, and execution model.

Term	Definition
Abstract Syntax Tree (AST)	A hierarchical, tree-shaped data structure that represents the grammatical structure of a GraphQL query after parsing. Each node in the tree corresponds to a construct in the query language (operation, field, argument, fragment). The AST serves as the intermediate representation between the raw query string and the execution engine.
Resolver	A function responsible for fetching or computing the data for a specific GraphQL field. When the execution engine encounters a field in the selection set, it calls the resolver function with four arguments: parent object, arguments, context, and execution info. Resolvers can be synchronous or asynchronous and may fetch data from databases, APIs, or other sources.
Introspection	In GraphQL, this refers to the built-in capability to query the schema itself. Clients can send introspection queries (using fields like `__schema`, `__type`) to discover what types, fields, and operations are available. In the database context, introspection refers to querying the database's system catalog (like `information_schema`) to discover tables, columns, and relationships.
Selection Set	The set of fields requested within a GraphQL operation or fragment. Represented as curly braces `{}` containing field selections, it defines the shape of the response data. Selection sets can be nested: each field that returns an object type may have its own selection set. The execution engine recursively processes selection sets to build the response tree.
Fragment	A reusable unit of GraphQL selection sets. **Named fragments** (`fragment UserFields on User { ... }`) are defined once and can be spread multiple times using `...UserFields`. **Inline fragments** (`... on User { ... }`) are anonymous and include a type condition. Fragments allow clients to avoid repeating complex selections and enable polymorphic queries on interfaces and unions.
Directive	An annotation in GraphQL queries or schemas that modifies execution behavior. Directives are prefixed with `@` and can have arguments. **Query directives** (like `@include`, `@skip`) control field inclusion at runtime. **Schema directives** (like `@deprecated`) provide metadata about schema elements. Custom directives can implement authorization, formatting, or other custom logic.
Operation	A single, named unit of work in a GraphQL document. There are three types: **query** (read operation), **mutation** (write operation), and **subscription** (real-time updates). Each document can contain multiple operations, but only one is executed per request (identified by the `operationName`).
Variable	A named placeholder in a GraphQL operation that gets replaced with values at execution time. Variables are defined in the operation header (`query($id: ID!)`) and referenced within the query body (`user(id: $id)`). This allows clients to parameterize queries without string concatenation.
Type Condition	The `on Type` part of a fragment that specifies which object type the fragment applies to. Inline fragments and fragment definitions must include a type condition. During execution, the fragment's selection set is only included if the runtime object type matches the type condition (for interfaces/unions) or is assignable to it.
Type System	The complete collection of GraphQL type definitions (`ObjectType`, `ScalarType`, `InterfaceType`, etc.) and the rules that govern their relationships and validity. The type system defines what queries are valid, what shape responses take, and how values are coerced. It serves as a contract between client and server.
Interface Implementation	The relationship where an `ObjectType` provides all the fields defined by an `InterfaceType`. An object type can implement multiple interfaces. The type system validates that implementing objects define all interface fields with compatible types. At runtime, interface fields are resolved by the concrete object's resolvers.
Type Modifier	A wrapper type that modifies another type's behavior. **`ListType`** indicates a list/array of values of the wrapped type. **`NonNullType`** indicates the wrapped type cannot be `null`. Modifiers can be nested (e.g., `[String!]!` means a non-null list of non-null strings).
Input Type	GraphQL types that can be used as arguments to fields or directives. Includes `ScalarType`, `EnumType`, `InputObjectType`, and modifiers thereof. Distinguished from output types because input types cannot have fields with arguments and cannot form cycles. The `is_input_type()` function checks this property.
Output Type	GraphQL types that can be returned by fields. Includes `ObjectType`, `InterfaceType`, `UnionType`, `ScalarType`, `EnumType`, and modifiers thereof. Output types can have fields with complex selection sets. The `is_output_type()` function checks this property.
Null Propagation	The GraphQL specification rule where if a non-null field (`String!`) resolves to `null`, that `null` propagates upward to the nearest nullable parent field, which becomes `null` in the response. This ensures type guarantees while allowing partial results. The `should_propagate_null()` function implements this logic.

Term	Definition
Parallel Execution	The execution engine optimization where independent sibling fields (fields at the same depth under the same parent) are resolved concurrently when possible. This improves performance for resolvers that fetch from different data sources. The engine uses async/await or threads to coordinate parallel execution.
Partial Results	GraphQL's approach to error handling where fields that resolve successfully are included in the response alongside errors from failed fields. Unlike REST, where an error typically aborts the entire response, GraphQL can return partial data with errors appended. The `ExecutionResult` contains both `data` and `errors` fields.
Schema Stitching	The technique of combining multiple GraphQL schemas (potentially from different services) into a single unified schema. The gateway pattern uses schema stitching to create a single entry point for clients while delegating to underlying services. This is a future extension point in our architecture.
Subscription	A GraphQL operation that establishes a long-lived connection (typically WebSocket) to receive real-time updates when events occur. Unlike queries/mutations that are request-response, subscriptions push data to clients. Our design includes `WebSocketTransport` as a future extension for subscriptions.
Custom Directives	User-defined schema directives that extend GraphQL execution with custom logic. Directives can be attached to schema elements (types, fields, arguments) and can modify runtime behavior through execution hooks. Our architecture includes `DirectiveMiddleware` for implementing custom directives.
Query Caching	Storing and reusing the results of GraphQL queries to improve performance. Can be implemented at multiple levels: HTTP response caching, parsed AST caching, compiled SQL caching, or database result caching. The `CacheBackend` abstract class provides an extension point for caching strategies.

Database Terminology

Relational database concepts essential for schema reflection and SQL compilation.

Term	Definition
Foreign Key	A database constraint that links a column or set of columns in one table (`child`) to the primary key columns of another table (`parent`). Foreign keys enforce referential integrity and define relationships between tables. During schema reflection, foreign keys are detected to generate GraphQL field relationships between types.
JOIN	A SQL operation that combines rows from two or more tables based on a related column between them. **INNER JOIN** returns rows with matching values in both tables. **LEFT JOIN** returns all rows from the left table with matching rows from the right (or NULLs if no match). The SQL compiler translates nested GraphQL fields into appropriate JOIN clauses.
Parameterization	The practice of using placeholders (`$1` , `?` , `%s`) in SQL statements instead of directly embedding values. Parameters are bound separately, preventing SQL injection attacks and allowing query plan reuse. Our `SQLBuilder` class automatically parameterizes values from GraphQL arguments in the `SQLParameter` structures.
N+1 Problem	A performance anti-pattern where an initial query fetches N parent records, then N additional queries (one per parent) fetch related child records. GraphQL naive execution often causes N+1 queries. Our engine prevents this through SQL JOIN compilation (single query) or DataLoader batching (batched queries).
Cartesian Product	The result of joining two or more tables without a join condition, producing all possible combinations of rows. Cartesian products explode result sizes (m × n rows) and indicate missing JOIN conditions. The SQL compiler must ensure every JOIN has an appropriate `ON` clause based on foreign keys.
LATERAL JOIN	A specialized SQL join (available in PostgreSQL, MySQL 8+) that allows a subquery in the JOIN to reference columns from preceding tables. Useful for efficiently resolving one-to-many relationships with limits/ordering per parent. The compiler may use LATERAL JOINs for nested collections with pagination.
Cursor Pagination	A pagination technique using opaque cursors (typically encoded row values) instead of numeric offsets. Cursors provide stable pagination across data changes (insertions/deletions). Our engine generates cursor-based pagination with `before` / `after` arguments that translate to SQL comparisons on indexed columns.
Result Mapper	A function that transforms flat SQL result rows (with joined columns from multiple tables) into the nested hierarchical structure expected by GraphQL. After executing a JOIN query, the result mapper walks the selection tree to reconstruct nested objects. It's part of the `SQLQuery` data structure.
Join Planning	The process of determining the optimal order and type of SQL JOINs to execute a GraphQL query. The planner analyzes relationship paths, estimates cardinalities, and avoids cartesian products. The `plan_joins()` function generates `JoinPlan` structures defining the join strategy.
System Catalog	Database-internal tables that store metadata about the database structure. In PostgreSQL: `pg_catalog` ; in SQL-standard databases: `information_schema` . Schema reflection queries these catalogs to discover tables, columns, constraints, and relationships.
Foreign Key Constraint	The formal database object that enforces a foreign key relationship. Contains source table/columns, target table/columns, and referential actions (`ON DELETE CASCADE`). The `RelationshipAnalyzer` examines these constraints to build `Relationship` metadata.
Cardinality	The numerical relationship between tables in a database. **One-to-one**: Each parent row relates to at most one child row. **One-to-many**: Each parent relates to many children. **Many-to-many**: Requires a junction table. The analyzer detects cardinality via unique constraints and foreign key directions.
Diamond Relationship	A query pattern where the same table is accessed via multiple paths in a single GraphQL query (e.g., `{ user { friends { posts }, posts } }` where `posts` appears through two paths). The SQL compiler must use table aliases to avoid column name conflicts when joining the same table multiple times.
Polymorphic Relationship	A database pattern where different rows in a table may reference different target tables (similar to GraphQL interfaces/unions). Implemented with a type discriminator column and multiple foreign key columns (only one populated per row). This maps naturally to GraphQL interface/union types.
Composite Primary Key	A primary key consisting of multiple columns rather than a single column. Common in junction tables and certain domain models. Schema reflection must handle composite keys when generating ID arguments and JOIN conditions (matching multiple columns).
Impedance Mismatch	The structural differences between GraphQL's hierarchical, graph-like data model and relational databases' tabular, set-based model. Our engine bridges this mismatch by translating nested selections to JOINs and hierarchical results to nested JSON.

Term	Definition
EXPLAIN Plan	Database command (`EXPLAIN SELECT ...`) that shows the query execution plan—how the database will execute the SQL, including index usage, join algorithms, and cost estimates. Our `SQLExplainer` class runs EXPLAIN on generated queries to diagnose performance issues.
Index Hint	A directive to the database optimizer suggesting which index to use for a query. Some databases support syntax like `USE INDEX (index_name)` . The compiler could add index hints based on GraphQL filter arguments to improve performance (future extension).
Stored Procedure	A precompiled collection of SQL statements stored in the database. Could be used as an alternative to dynamic SQL generation for common query patterns. Our architecture could extend to call stored procedures for certain GraphQL operations.
Transaction Isolation	Database property defining how concurrent transactions see each other's changes. Levels include READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE. The execution engine could wrap GraphQL mutations in transactions with appropriate isolation levels.
Connection Pool	A cache of database connections maintained so they can be reused, reducing connection establishment overhead. The execution engine would typically use a connection pool rather than creating new connections for each GraphQL request.

Compiler Terminology

Compiler concepts used in parsing, AST transformation, and query optimization.

Term	Definition
Lexer	Also called a **tokenizer**, this component breaks a source string (GraphQL query) into a sequence of **tokens**—the smallest meaningful units (keywords, identifiers, punctuation, literals). The `Tokenizer.tokenize()` method implements the lexer, producing `Token` objects with type, value, and location.
Parser	The component that analyzes the token sequence according to GraphQL grammar rules and builds an **Abstract Syntax Tree (AST)**. Our implementation uses **recursive descent parsing**, where each grammar rule corresponds to a function that may call itself recursively. The `Parser.parse_document()` method is the entry point.
Token	A single lexical unit from the source text, classified by `TokenType` (NAME, INT, STRING, PUNCTUATION, etc.). Each token has a `value` (the actual text) and `line`/`column` location for error reporting. The parser consumes tokens sequentially to construct the AST.
Intermediate Representation (IR)	A data structure that serves as an intermediate form between the source (GraphQL) and target (SQL). Our engine has multiple IRs: the AST (`Document`), execution plan, and `SQLQuery` structures. IRs are optimized and transformed before final code generation.
Optimization	The process of improving the efficiency of compiled code (SQL) without changing its semantics. Our SQL compiler performs optimizations like: eliminating unnecessary JOINs, pushing filters down, merging similar selections, and choosing optimal JOIN orders.
Traversal	The systematic visiting of nodes in a tree (AST) or graph. **Depth-first traversal** visits child nodes before siblings; **breadth-first** visits siblings before children. Our engine uses recursive traversal for: validating schemas, executing selection sets, and compiling selections to SQL.
Recursive Descent	A parsing technique where each non-terminal in the grammar corresponds to a function that may recursively call other non-terminal functions. Our parser uses recursive descent for its simplicity, error recovery, and good error messages—ideal for GraphQL's LL(1) grammar.
Lookahead	Examining upcoming tokens without consuming them, used to decide which parsing path to take. A **lookahead of k** means peeking at the next k tokens. GraphQL's grammar requires at most 1 token of lookahead (LL(1)), simplifying the parser implementation.
Syntax Error	An error in the query structure that violates GraphQL grammar rules (e.g., missing closing brace, unexpected token). The parser detects syntax errors and reports them with `GraphQLSyntaxError`, including `line` and `column` location from the `Token`.
Escape Sequence	In string literals, a backslash (`\`) followed by characters with special meaning: `\n` (newline), `\t` (tab), `\uXXXX` (Unicode code point). The lexer must recognize and properly decode escape sequences in `StringValue` tokens.
Block String	A triple-quoted string (`"""content"""`) in GraphQL that preserves newlines and allows unescaped quotes. Block strings have special whitespace handling: common leading indentation is removed. The lexer has a separate `TokenType.BLOCK_STRING` for these.
Fragment Spread	The syntax `...FragmentName` that references a previously defined named fragment. The parser must verify the fragment is defined and expand its selection set into the current location. Represented by the `FragmentSpread` AST node.
Inline Fragment	The syntax `... on Type { selection }` that defines an anonymous fragment with a type condition. Unlike fragment spreads, inline fragments don't have separate definitions. Represented by the `InlineFragment` AST node.
Backtracking	A parsing technique where the parser tentatively follows one production, and if it fails, rolls back to try an alternative. GraphQL's grammar is designed to avoid backtracking through careful token lookahead, making the parser more efficient and predictable.
Grammar Production	A rule in the formal grammar defining how symbols can be replaced. Example: `SelectionSet → { Selection* }`. Our parser implements each production as a method (`parse_selection_set()`) that consumes tokens according to the rule.
Left Recursion	A grammar pattern where a non-terminal appears as the first symbol of its own production (e.g., `Expression → Expression + Term`). Left recursion causes infinite loops in naive recursive descent parsers; GraphQL's grammar avoids left recursion.
Parse Tree	A detailed tree representation of how the grammar productions derive the input string. More detailed than AST, containing every syntactic detail. Our parser builds an AST (abstract) rather than a full parse tree, omitting irrelevant syntactic details.
Visitor Pattern	A design pattern for traversing complex structures (like ASTs) where operations are separated from structure traversal. A visitor interface has `visit_*` methods for each node type. Useful for validation, transformation, and code generation phases.

Term	Definition
Symbol Table	A data structure used by compilers to track information about identifiers (types, fields, variables). In our engine, the schema acts as a symbol table mapping type names to `GraphQLType` objects. Variable definitions are tracked during parsing/validation.
Static Single Assignment (SSA)	A compiler intermediate representation where each variable is assigned exactly once. Not used directly in our GraphQL compiler, but similar principles apply to ensuring field aliases and JOIN aliases are unique in generated SQL.
Code Generation	The final compiler phase that produces target code (SQL) from the optimized intermediate representation. Our `SQLBuilder` class handles code generation, producing parameterized SQL strings from `SQLQuery` structures.
Peephole Optimization	Local optimizations that examine small sequences of instructions (or SQL fragments) and replace them with more efficient sequences. Could be applied to generated SQL (e.g., simplifying `WHERE 1=1` or removing redundant conditions).
Constant Folding	Evaluating constant expressions at compile time rather than runtime. In GraphQL, this could involve evaluating literal arguments during compilation rather than passing them as SQL parameters (but must respect security boundaries).
Dead Code Elimination	Removing code that cannot be reached or whose results are never used. In our context, this could involve removing unused fields from selections (though GraphQL explicitly requests fields, so this is less applicable).
Inlining	Replacing a function call (or fragment spread) with the body of the function/fragment. Our parser performs fragment inlining early, expanding `FragmentSpread` nodes into their constituent selections during query preparation.
Abstract Syntax Tree (AST) Visualization	A human-readable display of the AST structure, useful for debugging. Our `ASTPrinter` class provides `print_document()`, `print_field()`, etc., methods that output indented text representations of the AST.
Resolving	In compilers, the process of mapping names to their definitions. In our engine: **type resolving** maps type names in the AST to `GraphQLType` objects; **field resolving** maps field names to `GraphQLField` definitions; **fragment resolving** maps fragment names to `FragmentDefinition` nodes.

Implementation Guidance

> **Implementation Note:** This glossary section doesn't require implementation code, but understanding these terms is critical for reading and implementing the other components. The tables above serve as a reference you can consult while working through the milestones.

Language-Specific Terminology Notes

- **Python-specific**: When implementing in Python, note that our type definitions use Python's `typing` module for type hints. The `@dataclass` decorator is commonly used for AST nodes and data structures. Async/await is used for asynchronous resolvers and parallel execution.
- **Go-specific** (if implementing in Go): Go's strong typing and interfaces align well with GraphQL's type system. Use structs for AST nodes with embedded types for inheritance-like behavior. Channels and goroutines enable parallel execution.
- **Rust-specific** (if implementing in Rust): Rust's enum types with associated data are ideal for representing AST nodes and GraphQL types. The ownership model ensures safe concurrent execution. Traits can define interfaces for resolvers and data loaders.

Terminology Usage Checklist

Before starting implementation, ensure you understand these key concept clusters:

1. **AST Construction**: Token → Lexer → Parser → Recursive Descent → Abstract Syntax Tree
2. **Type System**: Scalar/Object/Interface/Union/Enum → Input/Output Types → Type Modifiers (List/NonNull)
3. **Execution**: Resolver → Selection Set → Parallel Execution → Null Propagation → Partial Results
4. **Database Reflection**: System Catalog → Foreign Key → Cardinality → Type Mapping → Naming Convention
5. **SQL Compilation**: JOIN → Parameterization → N+1 Problem → Cartesian Product → Result Mapper

Quick Reference Table

For easy lookup during implementation, here's a condensed table of the most frequently used terms:

Category	Key Terms
AST Nodes	`Document`, `OperationDefinition`, `SelectionSet`, `Field`, `FragmentSpread`, `Argument`, `Directive`
Type System	`GraphQLType`, `ObjectType`, `ScalarType`, `InterfaceType`, `UnionType`, `EnumType`, `InputObjectType`
Execution	`Resolver`, `ExecutionContext`, `DataLoader`, `ExecutionResult`, `GraphQLError`, `PathSegment`
SQL Compilation	`SQLQuery`, `SQLSelect`, `SQLJoin`, `SQLParameter`, `JoinPlan`, `ResultMapper`
Database	`DatabaseMetadata`, `TableMetadata`, `ColumnMetadata`, `Relationship`, `ForeignKey`, `Cardinality`

Remember that all these terms map to concrete Python classes and functions defined in the Naming Conventions section. When you encounter an unfamiliar term in the codebase, refer back to this glossary for its definition and context.