# Extending sqlc: augmented generation of repositories in Go

Nikolay Kuznetsov

FOSDEM Brussels
1 Feb 2026

# About me

- Senior Software Engineer
- Zalando Helsinki 🇫🇮
- C → Java → Kotlin → **Go**
- Author of *pgx-outbox* and *sqlc++* projects

# Goals

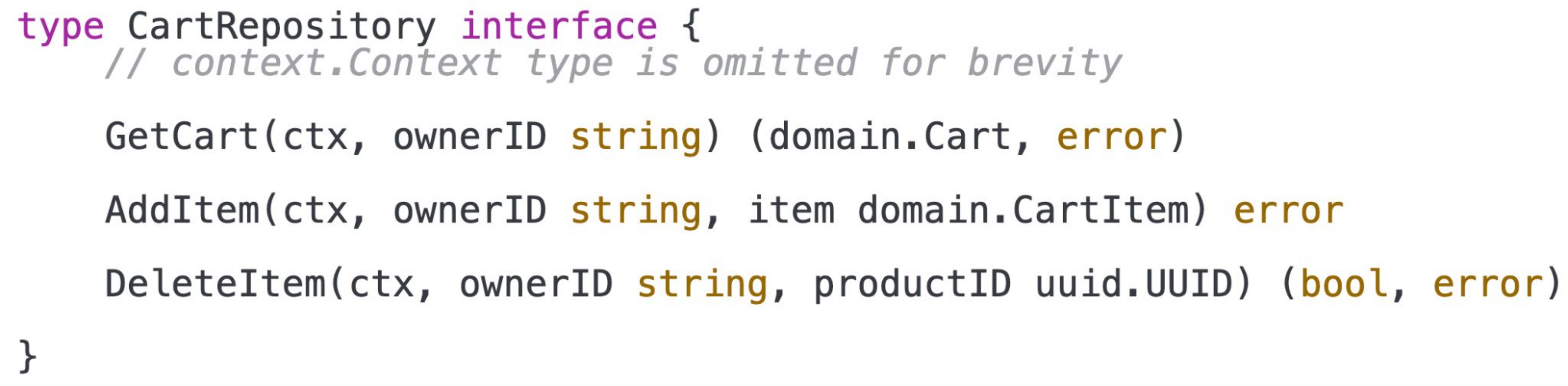
- Create repositories on top of *sqlc*-generated queries
- Keep strengths of *sqlc*: type safety & compile-time checks
- Decouple business logic from *sqlc*-generated records
- Automate repository creation with augmented generation

# Repository

- Accepts and returns domain models
- Hides details of SQL, schema, query libraries
- Maps between DB records and domain models
- Orchestrates transactions across multiple operations

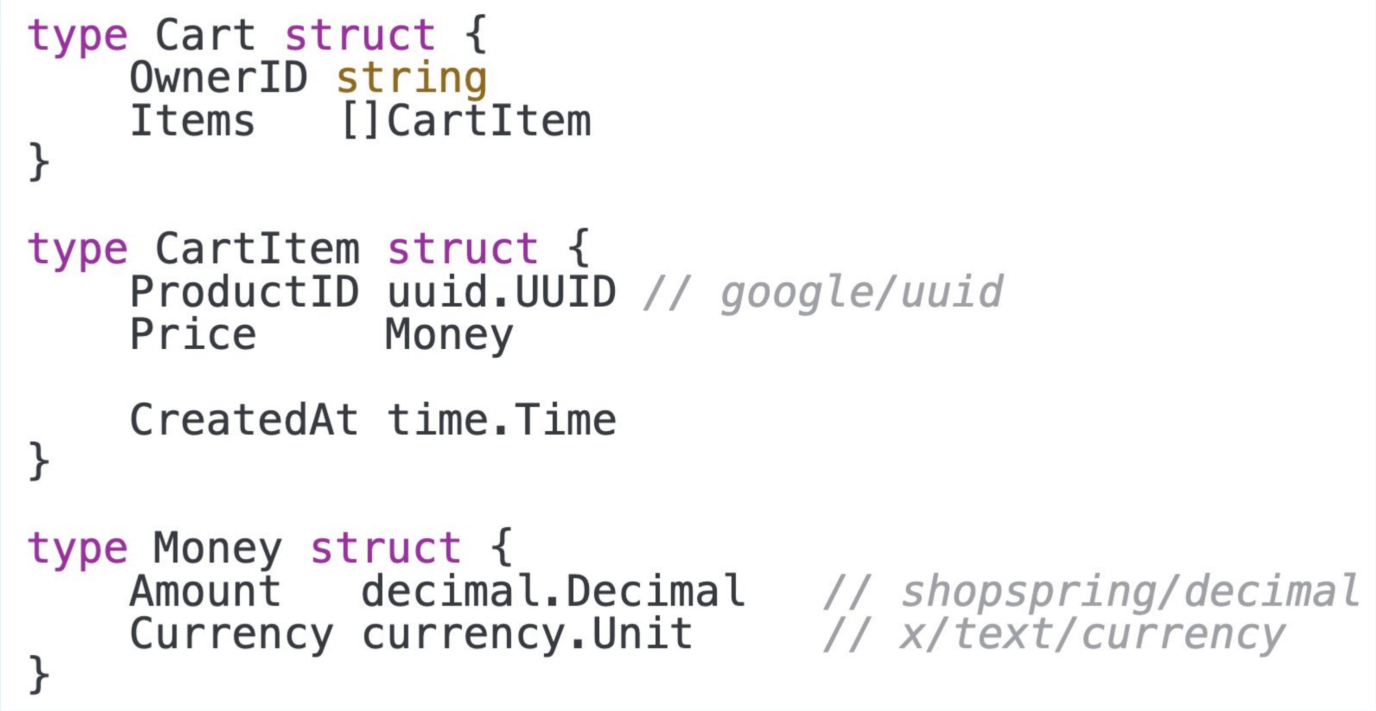
# Domain model

# Cart repository interface



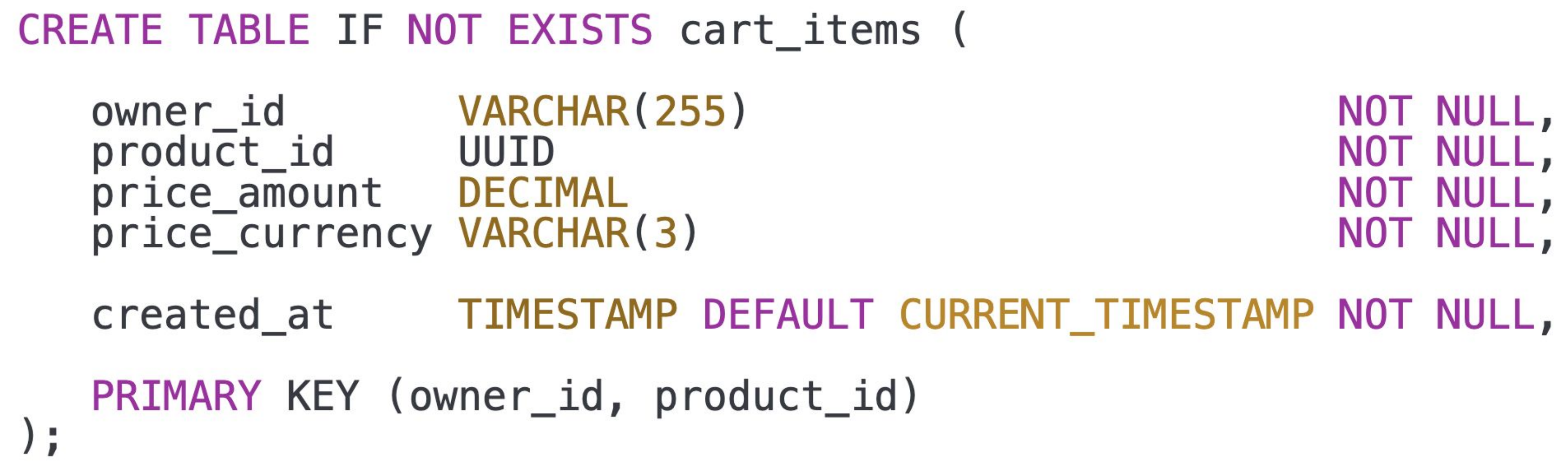
```
type CartRepository interface {
    // context.Context type is omitted for brevity

    GetCart(ctx, ownerID string) (domain.Cart, error)

    AddItem(ctx, ownerID string, item domain.CartItem) error

    DeleteItem(ctx, ownerID string, productID uuid.UUID) (bool, error)

}
```

# Domain cart



```go
type Cart struct {
    OwnerID string
    Items   []CartItem
}

type CartItem struct {
    ProductID uuid.UUID // google/uuid
    Price     Money

    CreatedAt time.Time
}

type Money struct {
    Amount   decimal.Decimal   // shopspring/decimal
    Currency currency.Unit     // x/text/currency
}
```

# Cart items table



```
CREATE TABLE IF NOT EXISTS cart_items (

   owner_id       VARCHAR(255)                        NOT NULL,
   product_id     UUID                                NOT NULL,
   price_amount   DECIMAL                             NOT NULL,
   price_currency VARCHAR(3)                          NOT NULL,

   created_at     TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,

   PRIMARY KEY (owner_id, product_id)
);
```

# Postgres in Go

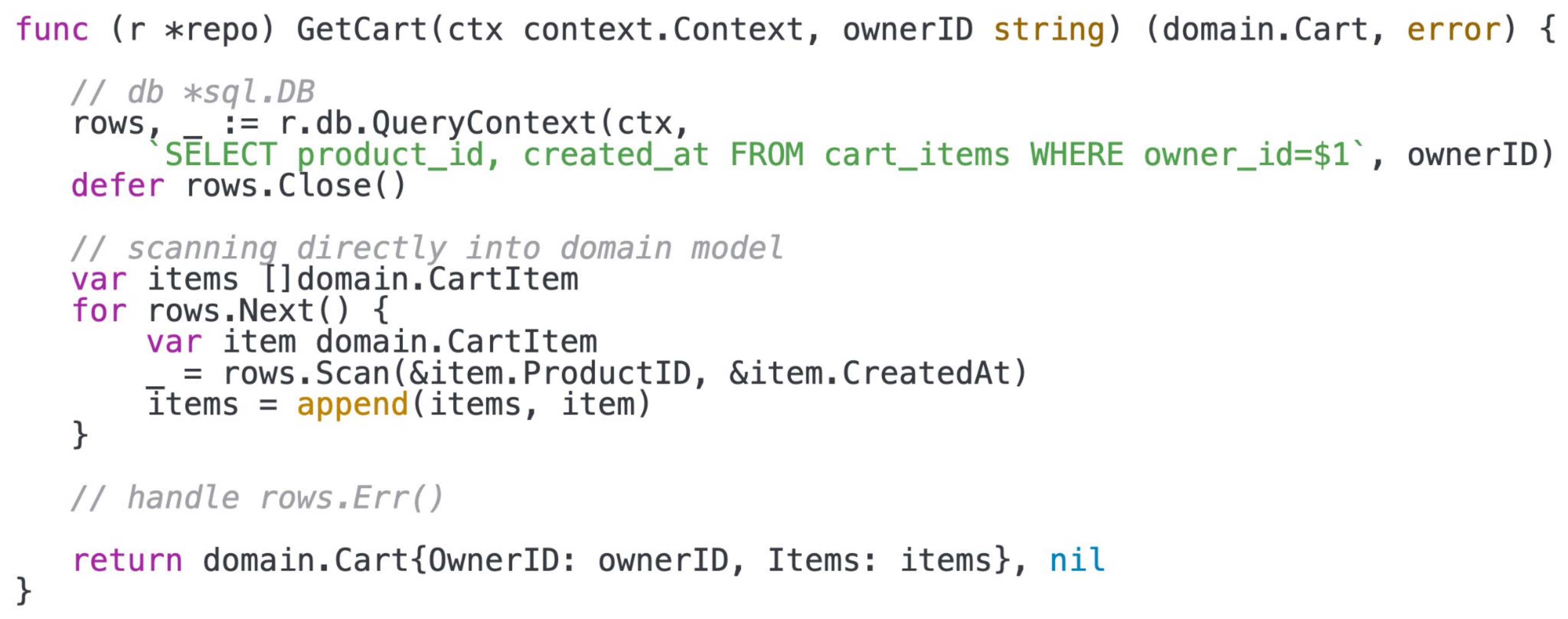
# Working with Postgres in Go

- Pure stdlib *database/sql*
- *squirrel* + *sqlx*
- *pgx* driver API
- ***sqlc***
- ~~ORMs~~ (*GORM*, *Ent*, *Bun*): out of scope

# Pure stdlib database/sql

- API: stdlib *database/sql*
- Driver: *lib/pq*
- Query builder: none
- Struct mapping: manual scanning

# Pure stdlib *database/sql* example

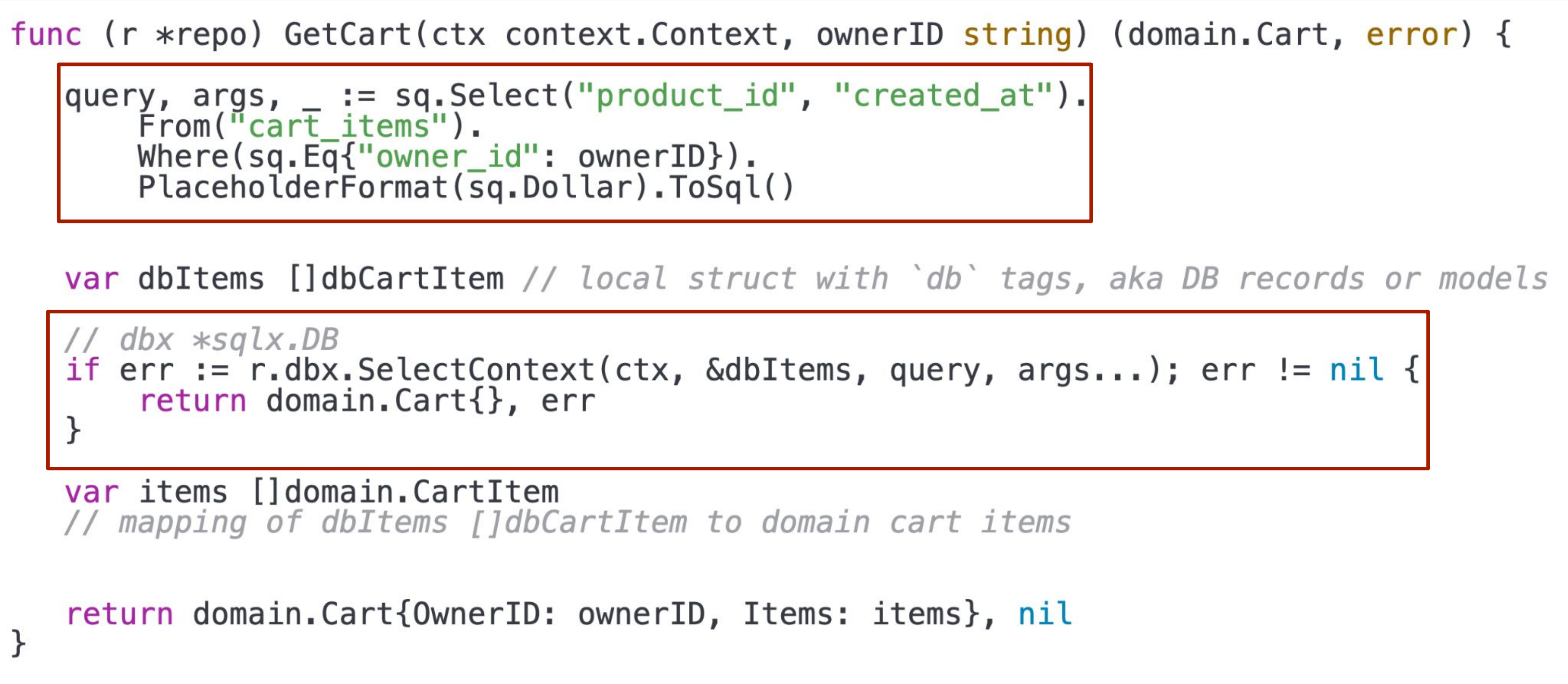


```go
func (r *repo) GetCart(ctx context.Context, ownerID string) (domain.Cart, error) {

    // db *sql.DB
    rows, _ := r.db.QueryContext(ctx,
        `SELECT product_id, created_at FROM cart_items WHERE owner_id=$1`, ownerID)
    defer rows.Close()

    // scanning directly into domain model
    var items []domain.CartItem
    for rows.Next() {
        var item domain.CartItem
        _ = rows.Scan(&item.ProductID, &item.CreatedAt)
        items = append(items, item)
    }

    // handle rows.Err()

    return domain.Cart{OwnerID: ownerID, Items: items}, nil
}
```

# Adopting *squirrel* and *sqlx*

- API: stdlib *database/sql*
- Driver: *lib/pq*
- Query builder: *Masterminds/squirrel*
- Struct mapping: *jmoiron/sqlx*

# Adopting *squirrel* and *sqlx* example

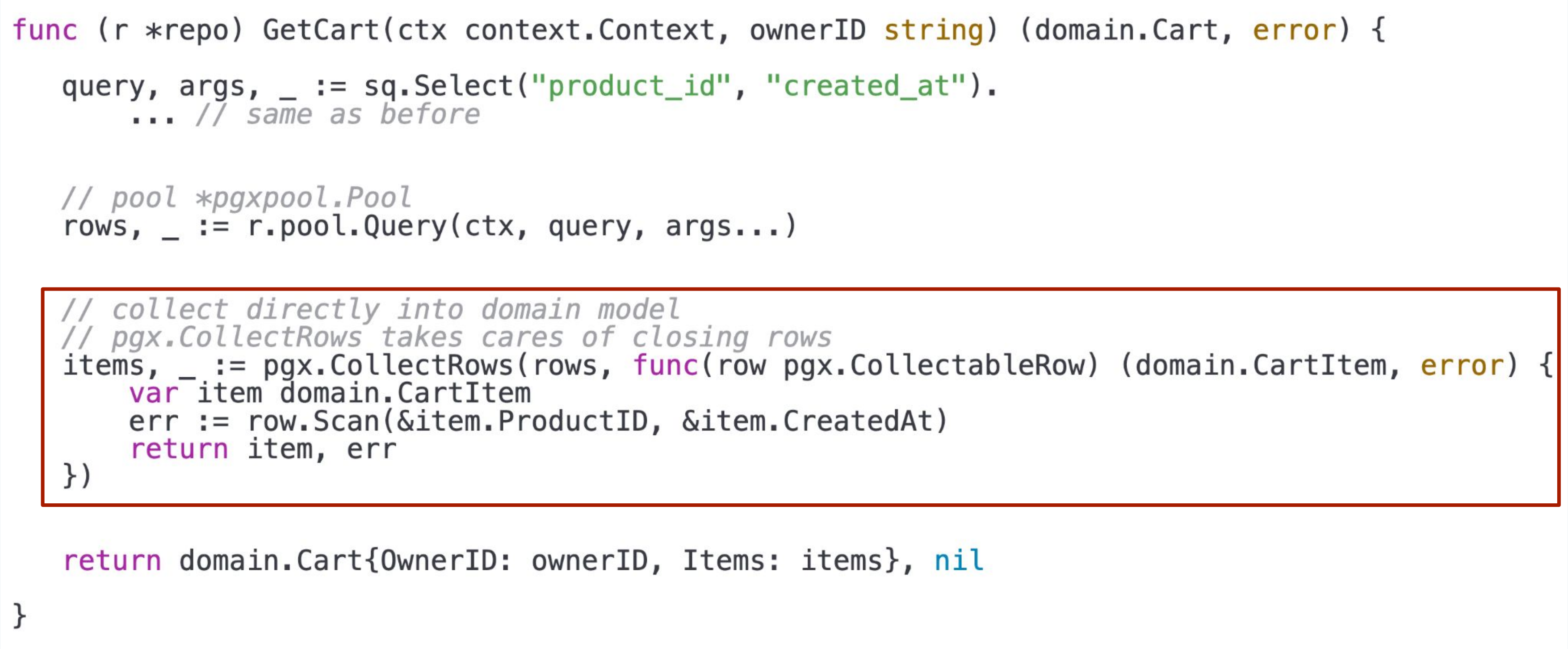


```
func (r *repo) GetCart(ctx context.Context, ownerID string) (domain.Cart, error) {
    query, args, _ := sq.Select("product_id", "created_at").
        From("cart_items").
        Where(sq.Eq{"owner_id": ownerID}).
        PlaceholderFormat(sq.Dollar).ToSql()

    var dbItems []dbCartItem // local struct with `db` tags, aka DB records or models

    // dbx *sqlx.DB
    if err := r.dbx.SelectContext(ctx, &dbItems, query, args...); err != nil {
        return domain.Cart{}, err
    }
    var items []domain.CartItem
    // mapping of dbItems []dbCartItem to domain cart items


    return domain.Cart{OwnerID: ownerID, Items: items}, nil
}
```

# Adopting *pgx*

- API: jackc/pgx (or *database/sql*)
- Driver: *jackc/pgx*
- Query builder: *Masterminds/squirrel*
- Struct mapping: *pgx.CollectRows* method

# Adopting *pgx* driver and API



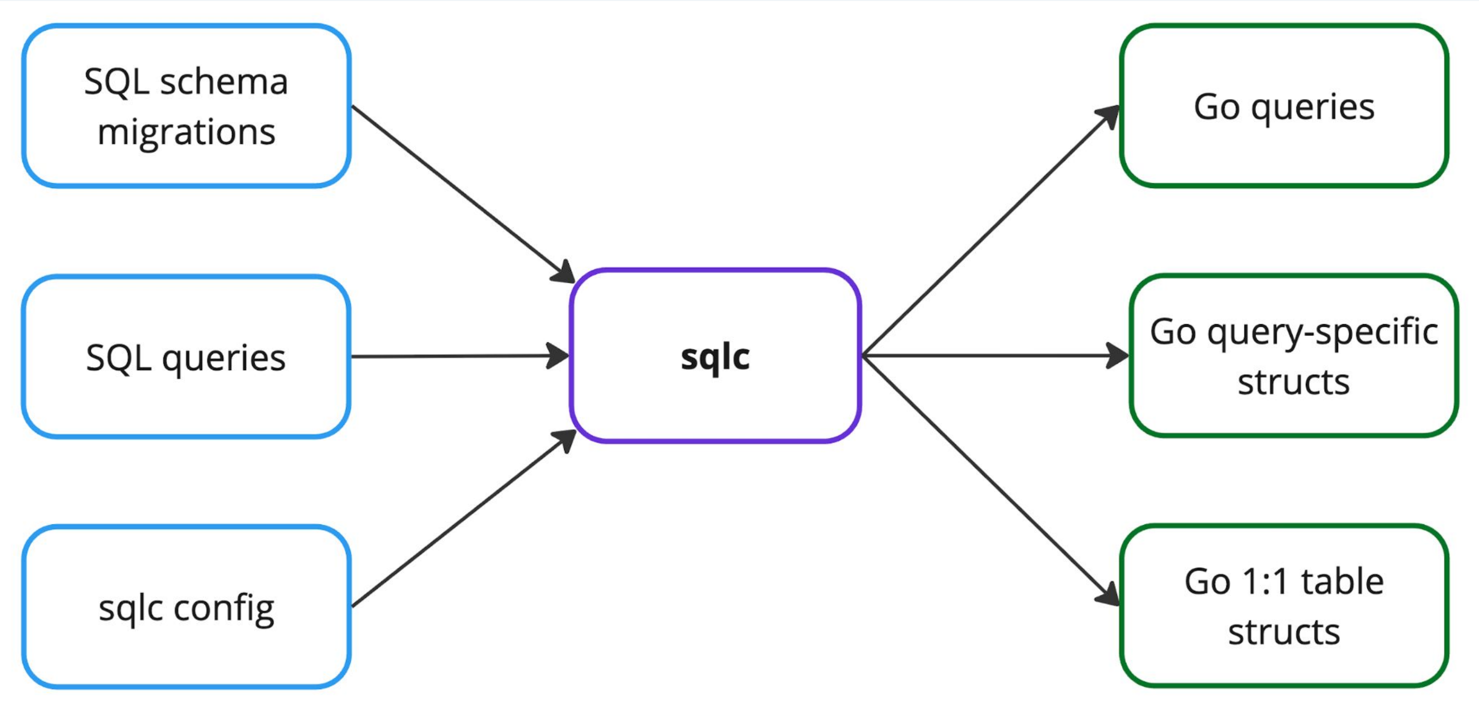
```
func (r *repo) GetCart(ctx context.Context, ownerID string) (domain.Cart, error) {

   query, args, _ := sq.Select("product_id", "created_at").
       ... // same as before


   // pool *pgxpool.Pool
   rows, _ := r.pool.Query(ctx, query, args...)


   // collect directly into domain model
   // pgx.CollectRows takes cares of closing rows
   items, _ := pgx.CollectRows(rows, func(row pgx.CollectableRow) (domain.CartItem, error) {
       var item domain.CartItem
       err := row.Scan(&item.ProductID, &item.CreatedAt)
       return item, err
   })


   return domain.Cart{OwnerID: ownerID, Items: items}, nil

}
```

# Pros of *pgx* API

- Full Postgres feature support
    - batching, LISTEN/NOTIFY, COPY, prepared statements
- Better type handling (JSON, arrays, UUID, etc)
- Advanced connection pool: *pgxpool*

sqlc

# sqlc in a nutshell



SQL schema migrations

SQL queries

sqlc config

**sqlc**

Go queries

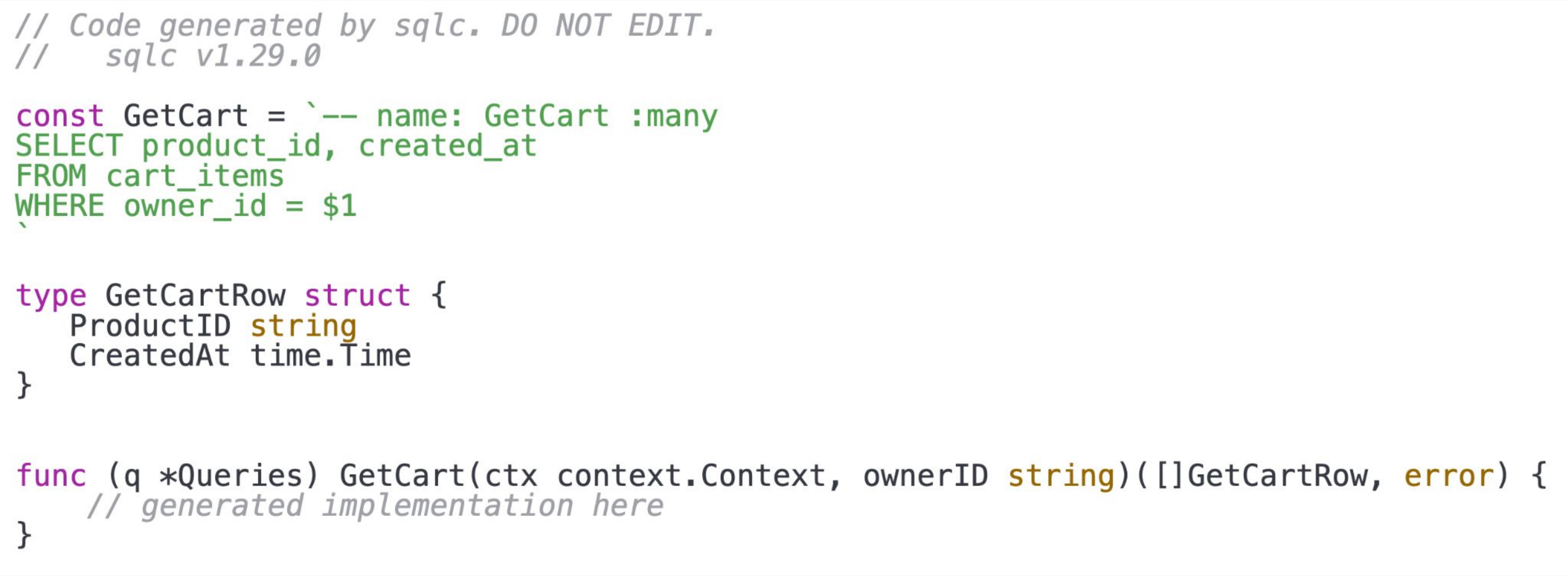
Go query-specific structs

Go 1:1 table structs

# Adopting *sqlc*

- API: *jackc/pgx*
- Driver: *jackc/pgx*
- Queries: pure SQL (almost)
- Struct mapping: automatic to *sqlc*-generated records

# Generated records and queries



```go
// Code generated by sqlc. DO NOT EDIT.
//   sqlc v1.29.0

const GetCart = `-- name: GetCart :many
SELECT product_id, created_at
FROM cart_items
WHERE owner_id = $1
`


type GetCartRow struct {
   ProductID string
   CreatedAt time.Time
}


func (q *Queries) GetCart(ctx context.Context, ownerID string)([]GetCartRow, error) {
    // generated implementation here
}
```

# Pros of *sqlc*

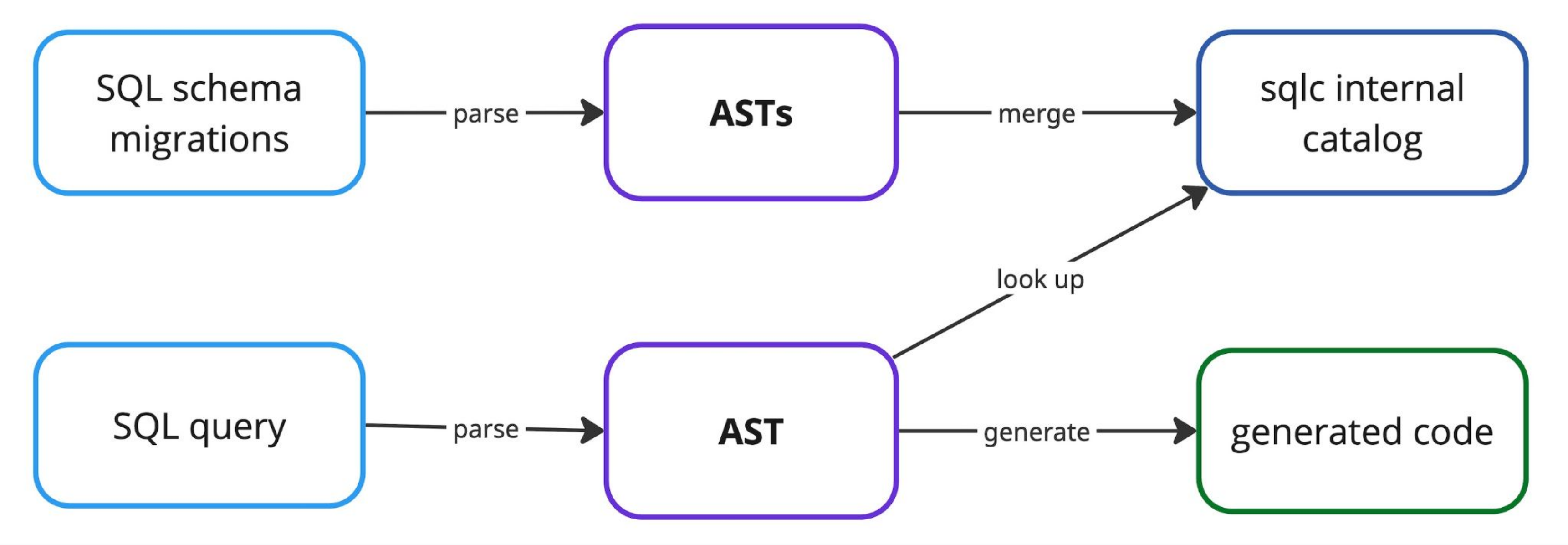
- Compile time schema and type safety
  - uses *Postgres* parser (*wasilibs/go-pgquery*) to produce *AST*
- Less boilerplate to write
  - query building, execution, rows scanning
- Separation of SQL and Go code

# Simplified Abstract Syntax Tree

```sql
-- name: DeleteItem :execrows
DELETE FROM cart_items WHERE owner_id = $1 AND product_id = $2;
```

```
DeleteStmt
 ├ Relation: cart_items
 └ WhereClause (BoolExpr: AND)
     ├ left: ColumnRef { Name: "owner_id" } = ParamRef{1}
     └ right: ColumnRef { Name: "product_id" } = ParamRef{2}
```

# sqlc under the hood


SQL schema migrations

parse

**ASTs**

merge

sqlc internal catalog

SQL query

parse

**AST**

look up

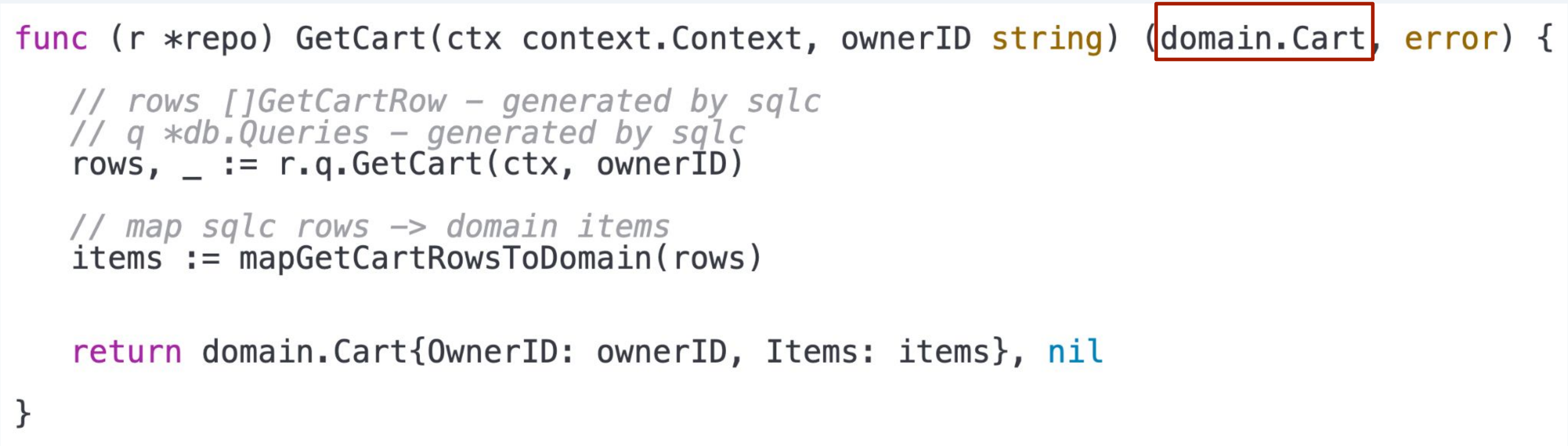
generate

generated code

# Challenges with *sqlc*

- Generated records leaking to business logic
- Mapping between records and domain models
- Dynamic conditional queries ✅
- Batch INSERTs and UPDATEs ✅

# Delegating to generated queries



```
func (r *repo) GetCart(ctx context.Context, ownerID string) (domain.Cart, error) {

    // rows []GetCartRow - generated by sqlc
    // q *db.Queries - generated by sqlc
    rows, _ := r.q.GetCart(ctx, ownerID)

    // map sqlc rows -> domain items
    items := mapGetCartRowsToDomain(rows)


    return domain.Cart{OwnerID: ownerID, Items: items}, nil

}
```

# Repository methods vs generated queries

- Repository methods align with **domain**, not SQL
- Repository methods compose multiple queries into a **transaction**
- Repository methods may add tracing, metrics, caching, etc

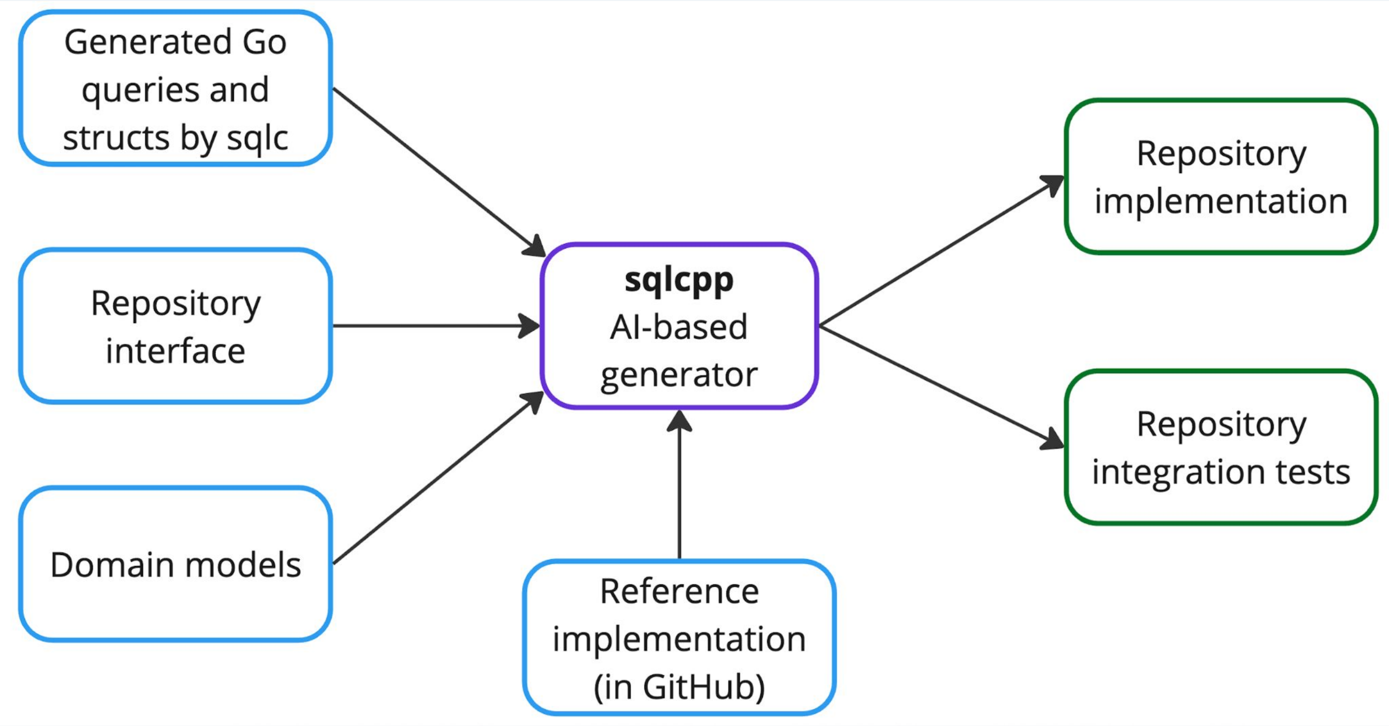
# sqlc++

# Introducing sqlc++

Let AI generate repository code by:

- Implementing provided interface (port)
- Using domain models and *sqlc*-generated artifacts
- Leveraging augmented generation

# sqlc++ in a nutshell



Generated Go queries and structs by sqlc

Repository interface

Domain models

**sqlcpp**
AI-based generator

Reference implementation (in GitHub)

Repository implementation

Repository integration tests

# sqlc++ variations

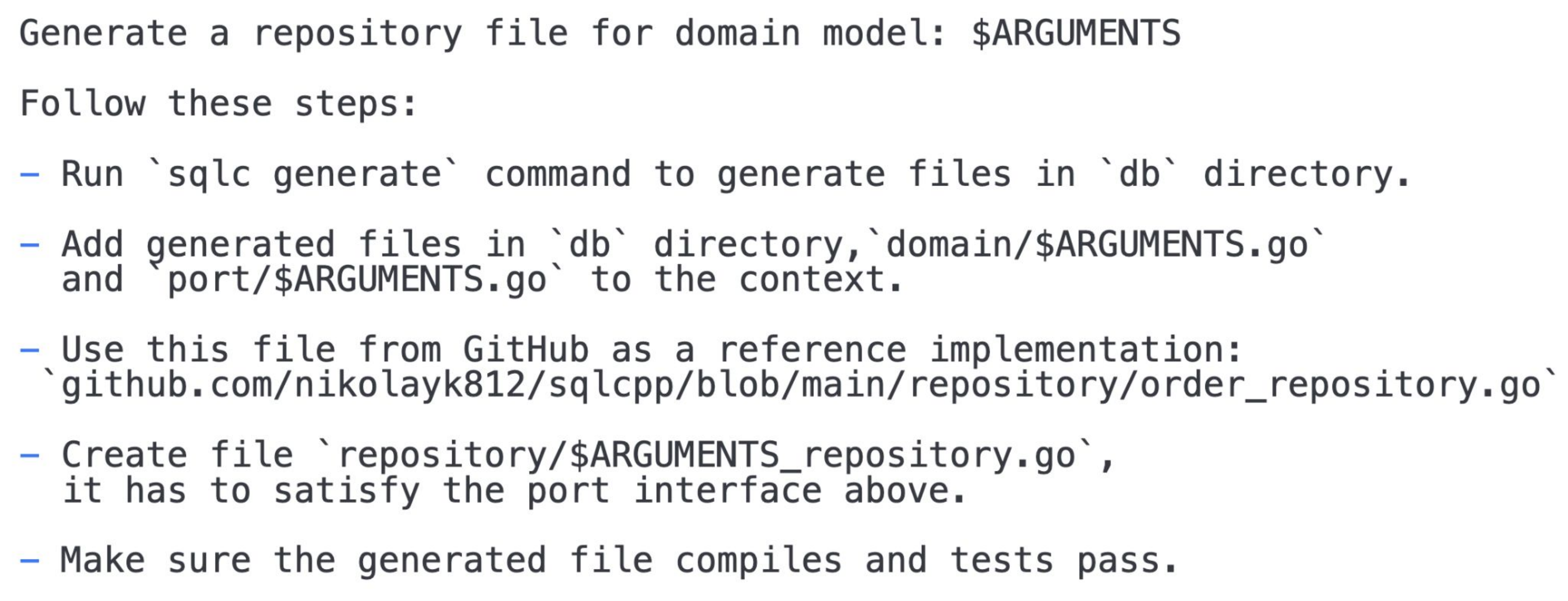
- Before agents: *langchain-go* & *text/template*
- **Agent with a custom command**
- Agent with skills

charm_
CRUSH

# Introducing Crush

- Glamorous coding agent
- Written in Go, open source (FSL)
- Multi-model, LSP-enhanced
- Agent skills, MCP, custom commands

# Custom command for Crush

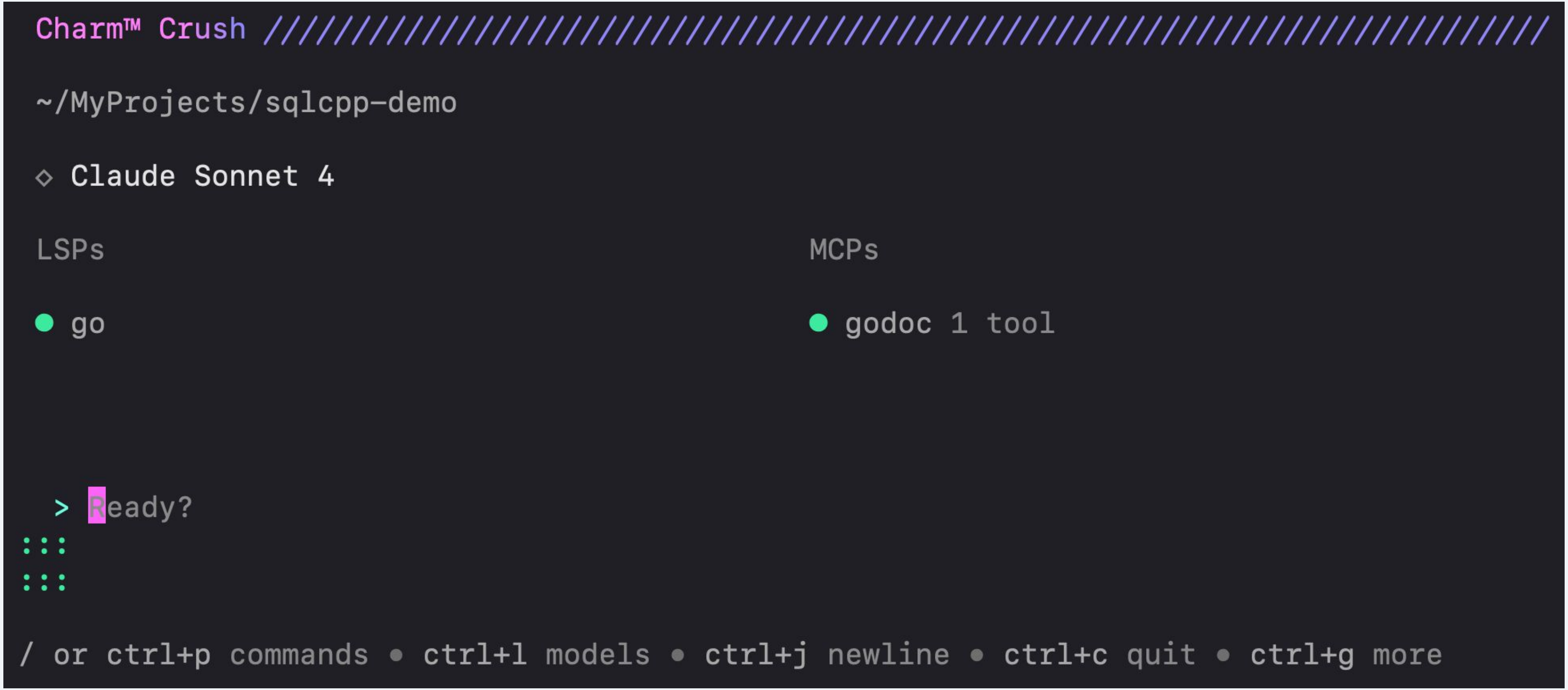


```
Generate a repository file for domain model: $ARGUMENTS

Follow these steps:

- Run `sqlc generate` command to generate files in `db` directory.

- Add generated files in `db` directory,`domain/$ARGUMENTS.go`
  and `port/$ARGUMENTS.go` to the context.

- Use this file from GitHub as a reference implementation:
 `github.com/nikolayk812/sqlcpp/blob/main/repository/order_repository.go`

- Create file `repository/$ARGUMENTS_repository.go`,
  it has to satisfy the port interface above.

- Make sure the generated file compiles and tests pass.
```

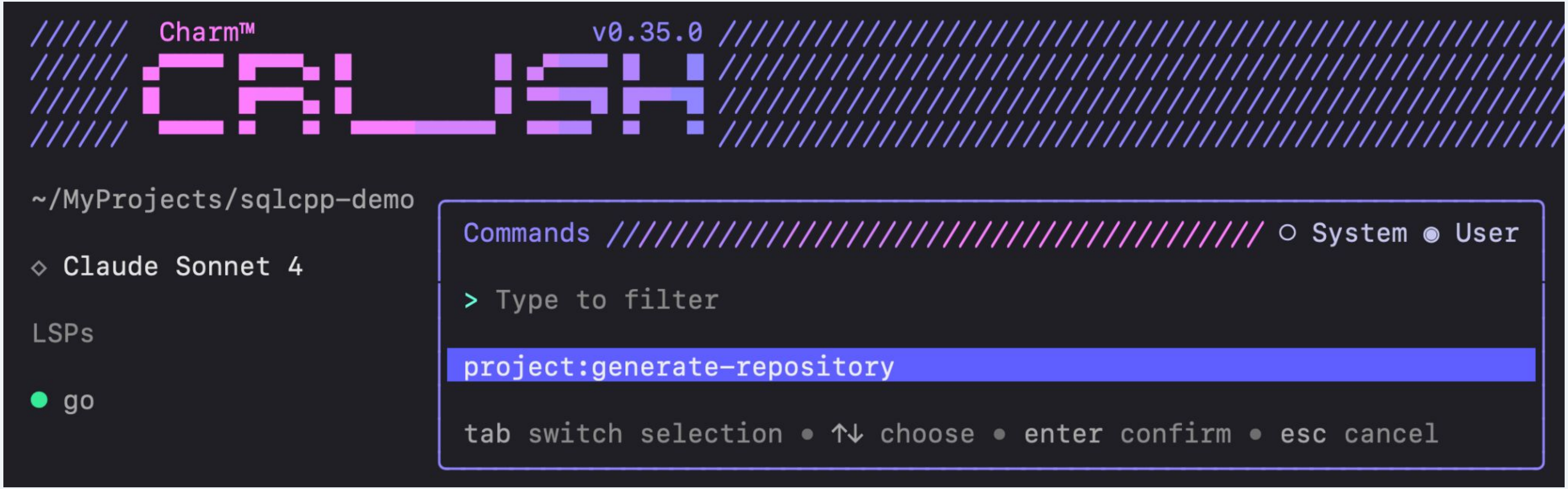
# Reference implementation project

- Domain model: order
- 2 tables: orders, order_items
- Various operations: CRUD, search by filter, soft-delete
- Various field types
- Transactions support

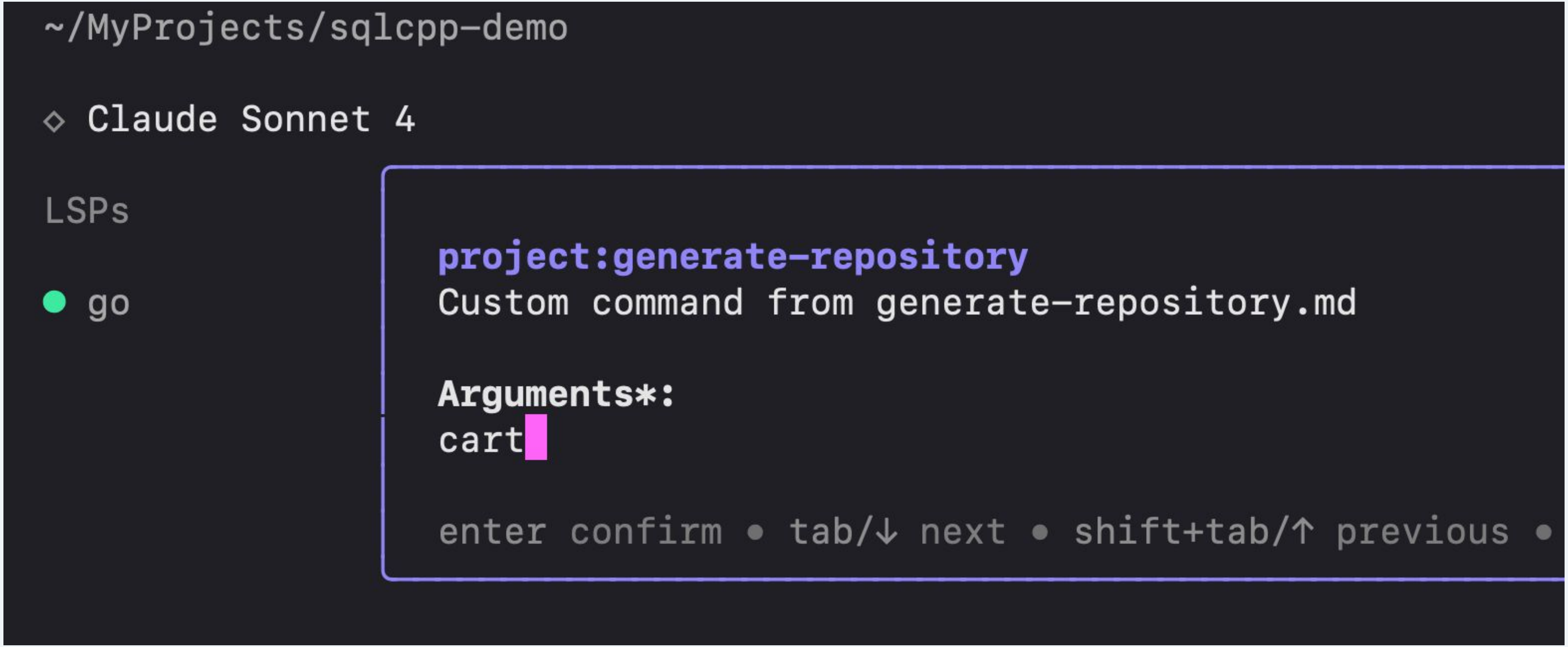
# Demo time



```
 Charm™ Crush /////////////////////////////////////////////////////////////////////////

 ~/MyProjects/sqlcpp-demo

 ◇ Claude Sonnet 4

 LSPs                                        MCPs

 ● go                                        ● godoc 1 tool



  > Ready?
:::
:::

/ or ctrl+p commands • ctrl+l models • ctrl+j newline • ctrl+c quit • ctrl+g more
```

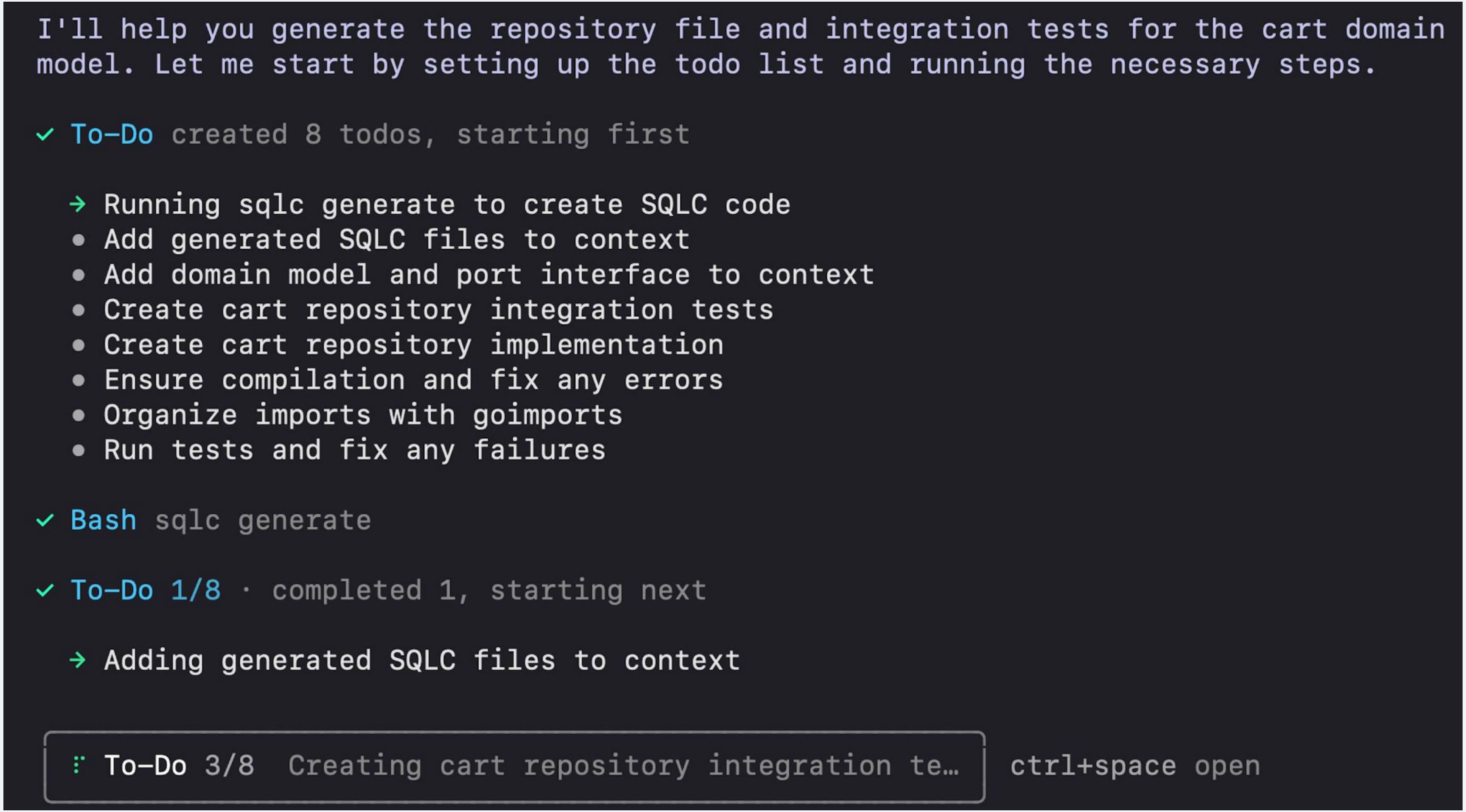


```
Charm™ v0.35.0
CRUSH

~/MyProjects/sqlcpp-demo     Commands //////////////////// ○ System ◉ User

◇ Claude Sonnet 4            > Type to filter

LSPs                         project:generate-repository

● go                         tab switch selection • ↑↓ choose • enter confirm • esc cancel
```

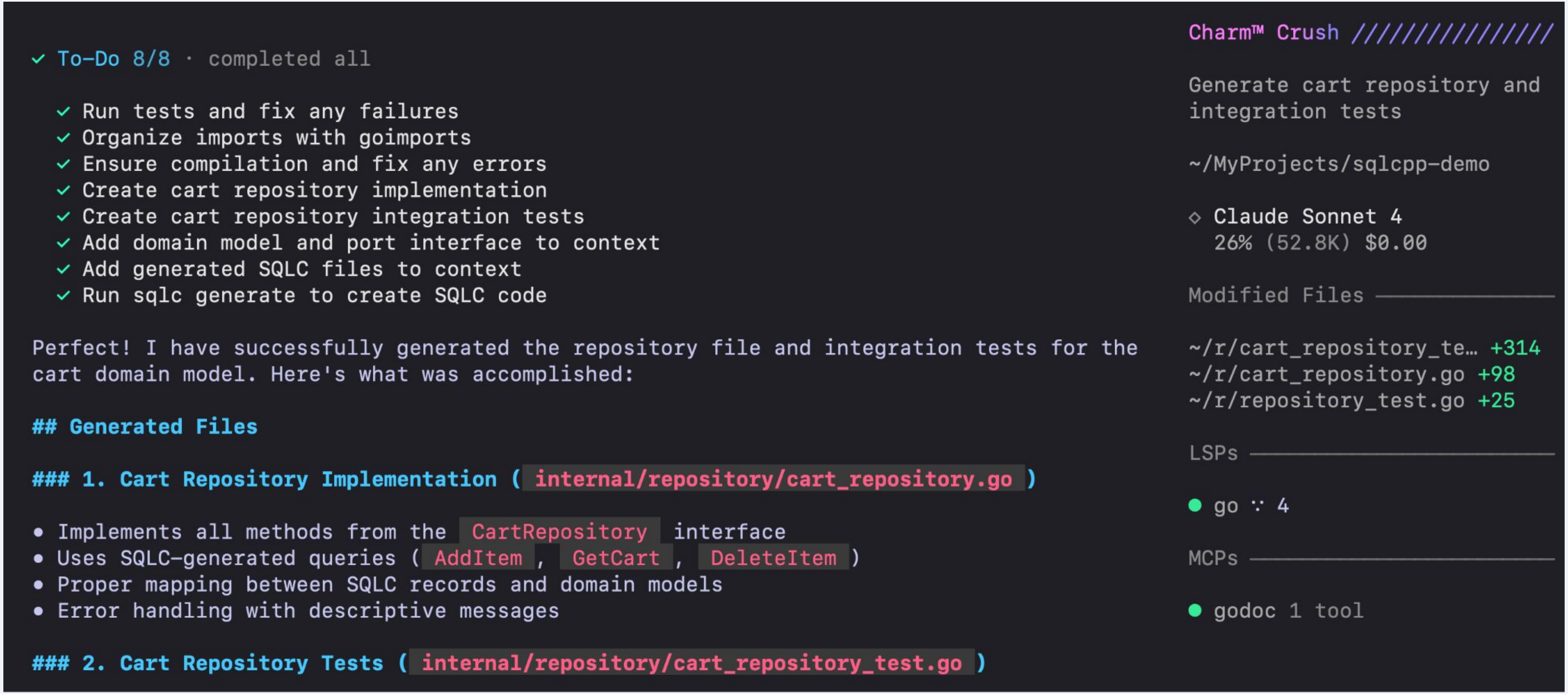


```
~/MyProjects/sqlcpp-demo

◇ Claude Sonnet 4

LSPs

● go
```

```
project:generate-repository
Custom command from generate-repository.md

Arguments*:
cart

enter confirm • tab/↓ next • shift+tab/↑ previous •
```



```
I'll help you generate the repository file and integration tests for the cart domain
model. Let me start by setting up the todo list and running the necessary steps.

✓ To-Do created 8 todos, starting first

  → Running sqlc generate to create SQLC code
  • Add generated SQLC files to context
  • Add domain model and port interface to context
  • Create cart repository integration tests
  • Create cart repository implementation
  • Ensure compilation and fix any errors
  • Organize imports with goimports
  • Run tests and fix any failures

✓ Bash sqlc generate

✓ To-Do 1/8 · completed 1, starting next

  → Adding generated SQLC files to context


 ⁝ To-Do 3/8  Creating cart repository integration te…   ctrl+space open
```



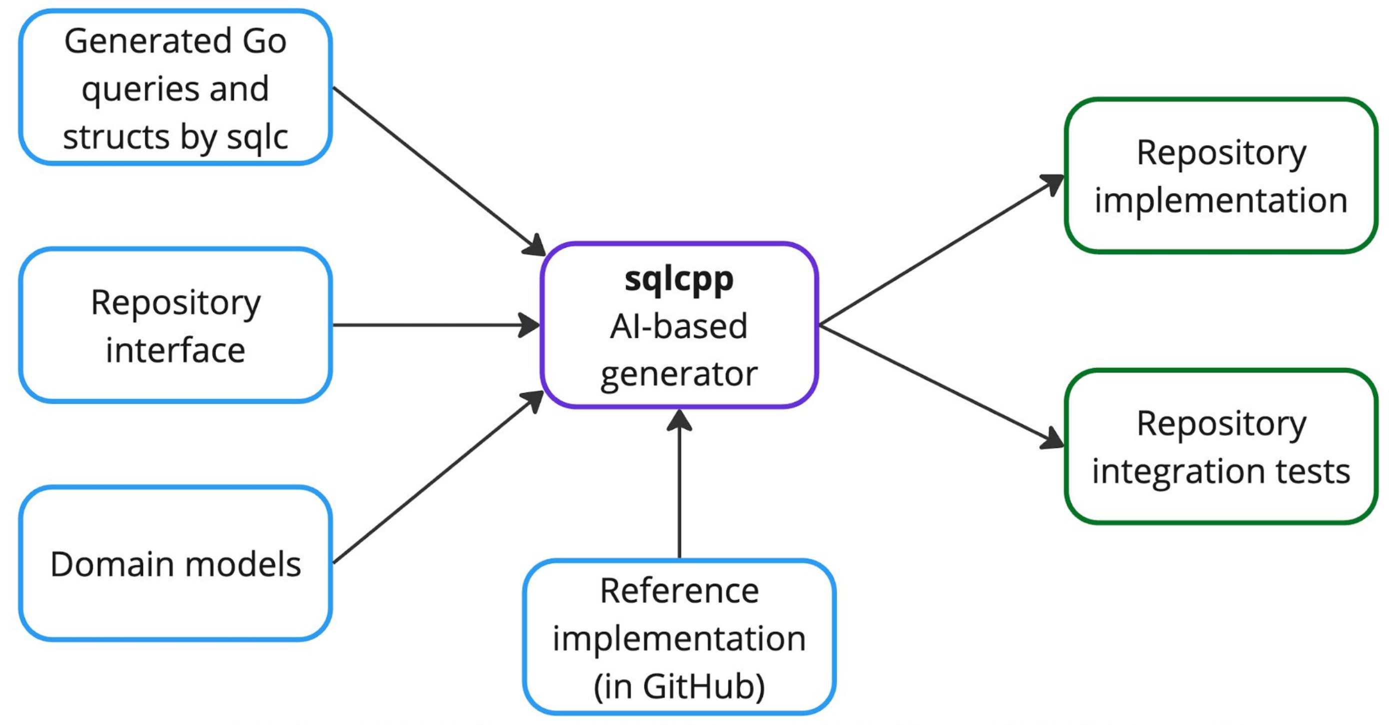
```
✓ To-Do 8/8 · completed all

  ✓ Run tests and fix any failures
  ✓ Organize imports with goimports
  ✓ Ensure compilation and fix any errors
  ✓ Create cart repository implementation
  ✓ Create cart repository integration tests
  ✓ Add domain model and port interface to context
  ✓ Add generated SQLC files to context
  ✓ Run sqlc generate to create SQLC code

Perfect! I have successfully generated the repository file and integration tests for the
cart domain model. Here's what was accomplished:

## Generated Files

### 1. Cart Repository Implementation ( internal/repository/cart_repository.go )

• Implements all methods from the  CartRepository  interface
• Uses SQLC-generated queries ( AddItem ,  GetCart ,  DeleteItem )
• Proper mapping between SQLC records and domain models
• Error handling with descriptive messages

### 2. Cart Repository Tests ( internal/repository/cart_repository_test.go )
```

```
Charm™ Crush ///////////////

Generate cart repository and
integration tests

~/MyProjects/sqlcpp-demo

◇ Claude Sonnet 4
  26% (52.8K) $0.00

Modified Files ──────────────

~/r/cart_repository_te… +314
~/r/cart_repository.go +98
~/r/repository_test.go +25

LSPs ────────────────────────

● go ∵ 4

MCPs ────────────────────────

● godoc 1 tool
```

# Testing repositories

- Testcontainers + Postgres module
- *Suite* from *stretchr/testify*
- Table tests for each repository method
- Helpful libs: *gofakeit*, *go-cmp*

# sqlc++ architecture recap


Generated Go queries and structs by sqlc
Repository interface
Domain models
**sqlcpp**
AI-based generator
Reference implementation (in GitHub)
Repository implementation
Repository integration tests

# Takeaways

- Adopt ***sqlc*** and ***Crush***, contribute
- Avoid *sqlc*-generated structs in business logic
- Repository boilerplate can be automated
- Augmented generation provides more stable results

# Thank you!

# Q&A

# Nikolay Kuznetsov

nikolayk812

nkuznetsov

github.com/nikolayk812/sqlcpp