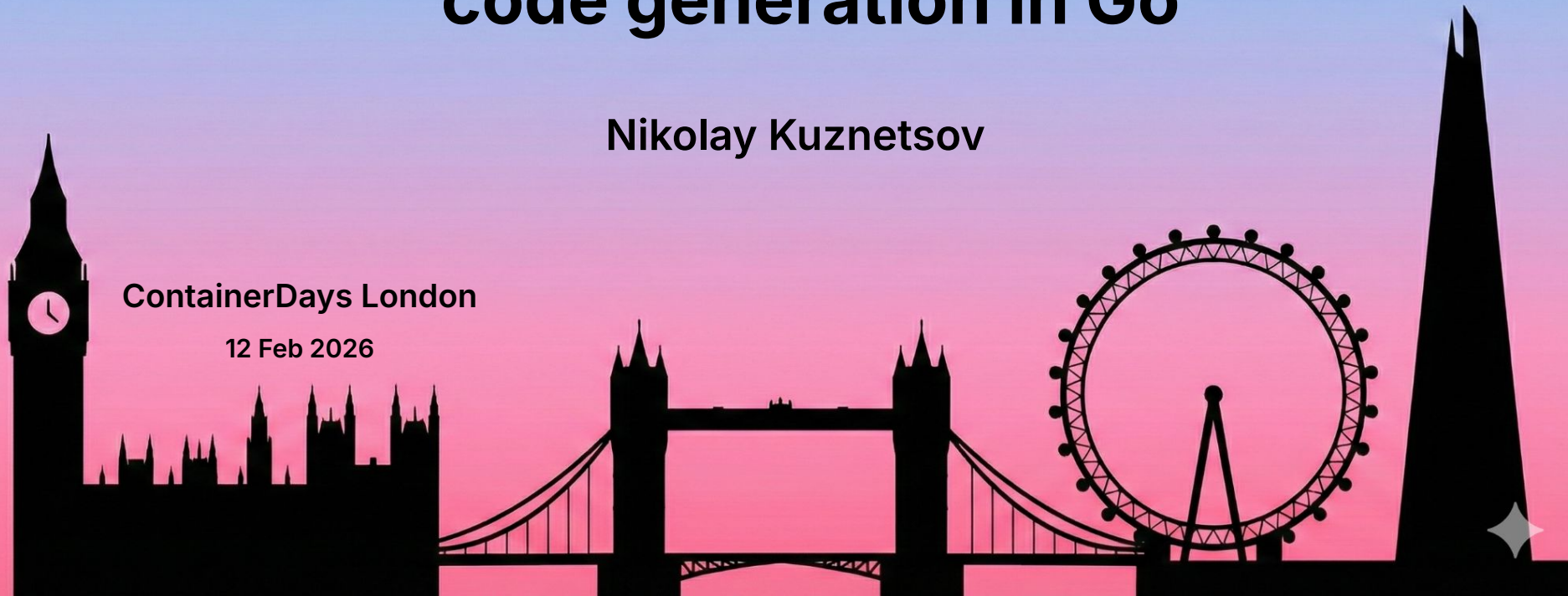
Practical sqlc: leveraging SQL-first code generation in Go

Nikolay Kuznetsov



ContainerDays London

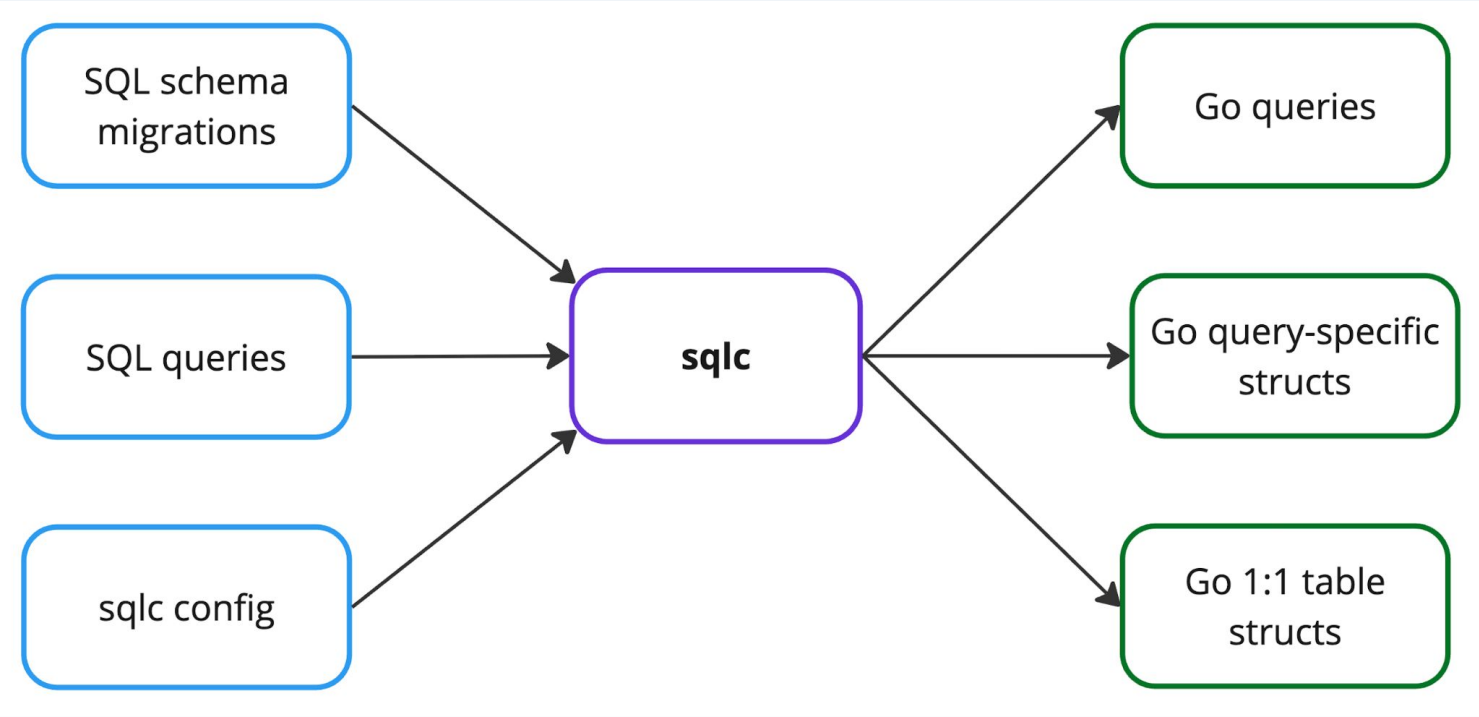
12 Feb 2026

About me

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


sqlc

sqlc in a nutshell



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)
);
```

Generated records and queries

```
// 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
}
```

sqlc minimal config

```
version: "2"
sql:
  - engine: "postgresql"
    schema: "internal/migrations"
    queries: "internal/db/queries"
    gen:
      go:
        package: "db"
        out: "internal/db"
        sql_package: "pgx/v5"
```

Pros of *sqlc*

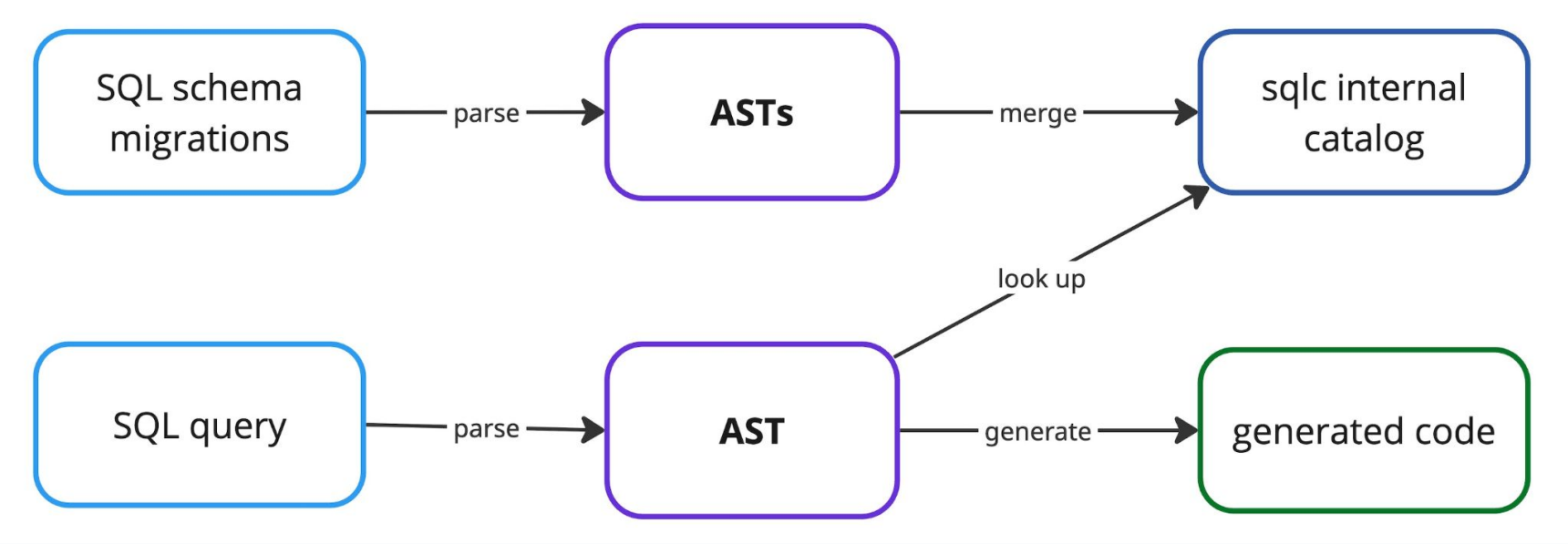
- Compile time schema and type safety
  - uses *Postgres* parser (*wasilibs/go-pgquery*) to produce ***AST***
- Less boilerplate code to write
  - query building, execution, rows scanning
- Separation of SQL and Go code

Simplified Abstract Syntax Tree

```
-- 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



sqlc demo

Postgres in Go recap

Working with Postgres in Go

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

Domain cart

```
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
}
```

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

```go
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

```go
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*

Summary of approaches

	Query building	Query execution	Rows scanning	Domain models mapping
Pure stdlib database/sql	manual	sql.DB	manual: rows.Next()	in rows scanning
squirrel + sqlx	dynamic in Go	sqlx.DB	Automatic to sqlx-records	manual
pgxpool API (+ squirrel)	dynamic in Go	pgxpool / DBTX	manual: pgx.CollectRows()	in rows scanning
sqlc	**static** in SQL files	DBTX	**Automatic** to generated records	**manual**

Challenges with sqlc

Potential challenges with *sqlc*

- Dynamic conditional queries
- Generated Go-types choice
- Generated records leaking to business logic
- Batch operations: INSERTs and UPDATEs
- Transaction support

- Dynamic query challenge

- Use case: find all cart items to satisfy the filter

```
type CartFilter struct {
    // optional filter by owner id
    OwnerID    *string

    // optional filter by product ids
    // (OR semantics)
    ProductIDs []uuid.UUID
}
```

Dynamic query options

- Keep SQL and *sqlc*, leverage short-circuiting
- Fallback to Go and *squirrel*
  - for a specific query only
  - implement in repository layer

Short-circuiting

```sql
-- name: SearchCartItems :many
SELECT *
FROM cart_items
WHERE
  -- short-circuiting: entire condition is TRUE
  -- (filter ignored) when product_ids are NULL
    (@product_ids::UUID[] IS NULL
          OR product_id = ANY(@product_ids))

  -- sqlc.narg() macro generates *string in
  -- SearchCartItemsParams struct
  AND (sqlc.narg(owner_id)::VARCHAR IS NULL
          OR owner_id = sqlc.narg(owner_id));
```

- Types choice challenge

- Default: most accurate database representation: *pgtypes*
- Types can be overridden in *sqlc.yaml*
- Rationale: generate types similar to those used in domain

```
type GetCartRow struct {
    ProductID     pgtype.UUID
    PriceAmount   pgtype.Numeric
    PriceCurrency string
    CreatedAt     pgtype.Timestamp
}
```

```
type GetCartRow struct {
    ProductID     uuid.UUID
    PriceAmount   decimal.Decimal
    PriceCurrency string
    CreatedAt     time.Time
}
```

Type overrides

```
overrides:
  - db_type: "uuid"
    go_type:
      import: "github.com/google/uuid"
      type: "UUID"
  - db_type: "pg_catalog.timestamp"
    go_type:
      import: "time"
      type: "Time"
  - db_type: "pg_catalog.numeric"
    go_type:
      import: "github.com/shopspring/decimal"
      type: "Decimal"
```

- Batch operations challenge

- Use case: INSERT / UPDATE multiple rows efficiently
- Leverage ***:batchexec*** and ***:batchmany*** query annotations
- *pgx.Batch* gets generated
- *pgx* uses prepared statements and Postgres pipeline mode

Batch INSERT example

```
-- name: AddItemsBatch :batchexec
INSERT INTO cart_items
    (owner_id, product_id, price_amount, price_currency)
VALUES ($1, $2, $3, $4)
ON CONFLICT (owner_id, product_id) DO NOTHING;
```

Generated pgx.Batch code

```
batch := &pgx.Batch{}

// arg []AddItemsBatchParams
for _, a := range arg {
    // set vals from a
    batch.Queue(AddItemsBatch, vals...)
}

// pgx.BatchResults
return q.db.SendBatch(ctx, batch)
```

- Repository challenge

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

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)
}
```

Delegating to generated queries

```go
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

}
```

- Transaction support challenge

- Use case: run multiple *sqlc*-queries in a tx
- Use case: run multiple repo methods in a tx
- Leverage *DTBX* interface generated by sqlc
- *withTx* and exported *RunWithTx* helpers

DBTX interface

```
// common interface between pgxpool.Pool and pgx.Tx
type DBTX interface {
    Exec(ctx, string, ...interface{}) (pgconn.CommandTag, error)
    Query(ctx, string, ...interface{}) (pgx.Rows, error)
    QueryRow(ctx, string, ...interface{}) pgx.Row
    // only if batch annotations are used
    SendBatch(ctx, *pgx.Batch) pgx.BatchResults
}
```

withTx helper

```
err := withTx(ctx, r.dbtx, func(q *db.Queries) error {

    if err := q.AddItem(ctx, addParams); err != nil {
        return err
    }

    if err := q.DeleteItem(ctx, deleteParams); err != nil {
        return err
    }

    return nil
})
```

withTx internals

```go
// withTx executes fn within a transaction if dbtx is a pool,
// or uses the existing transaction if dbtx is already a transaction.

func withTx(ctx, dbtx db.DBTX, fn func(q *db.Queries) error) error {

    if tx, ok := dbtx.(pgx.Tx); ok {
        return fn(db.New(tx))
    }

    // pool, ok := dbtx.(*pgxpool.Pool)
    // pool.Begin, fn(db.New(tx)), tx.Commit/Rollback logic
```

Exported *RunWithTx* helper

- Service layer can orchestrate multiple methods
- Service layer does NOT import *pgx* driver internals (tx)

```
func (r *cartRepo) RunWithTx(ctx, fn func(port.CartRepository) error) error {

    return withTx(ctx, r.q.DBTX(), func(txQ *db.Queries) error {
        txRepo := &cartRepo{q: txQ}
        return fn(txRepo)
    })

}
```

sqlc++

Goals

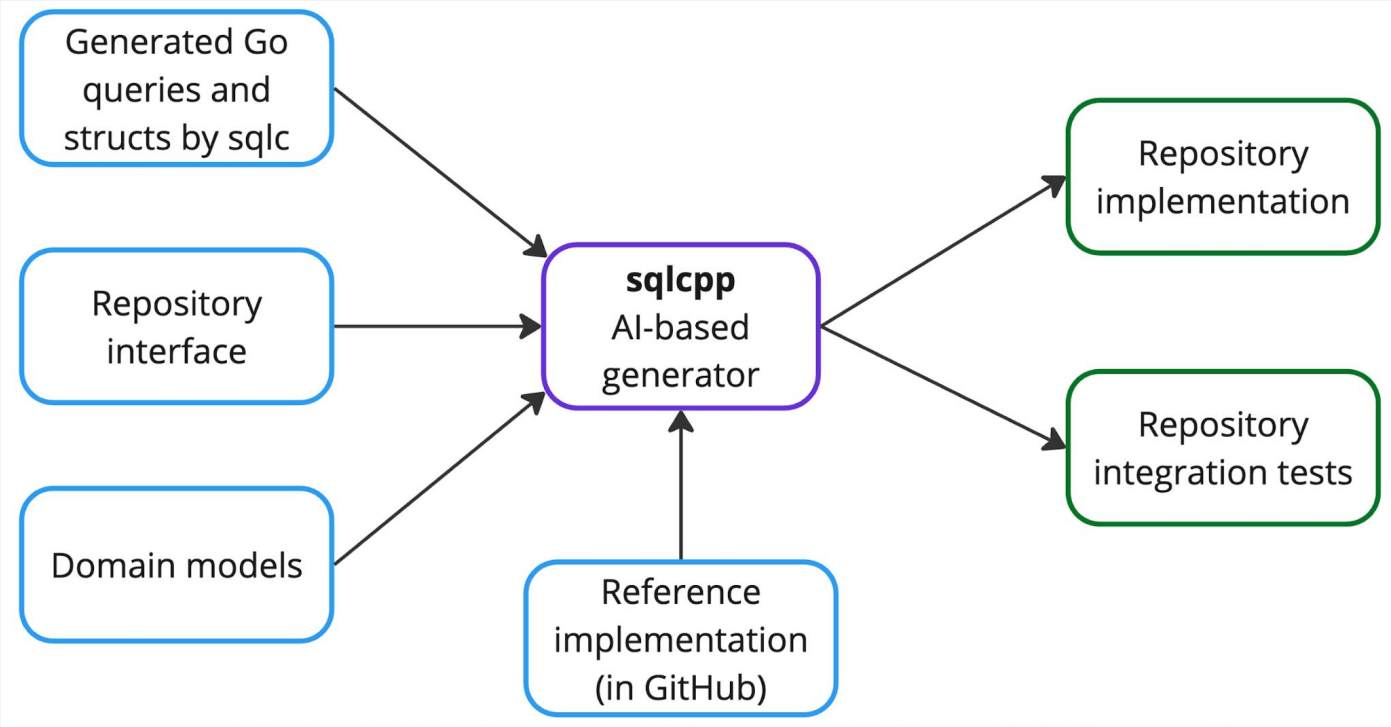
- Create repositories on top of *sqlc*-generated queries
- Keep strengths of *sqlc*: type safety & compile-time checks
- Decouple business logic from *sqlc*-generated records
- Automate repository creation with augmented generation

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



sqlc++ generations

- 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.
```

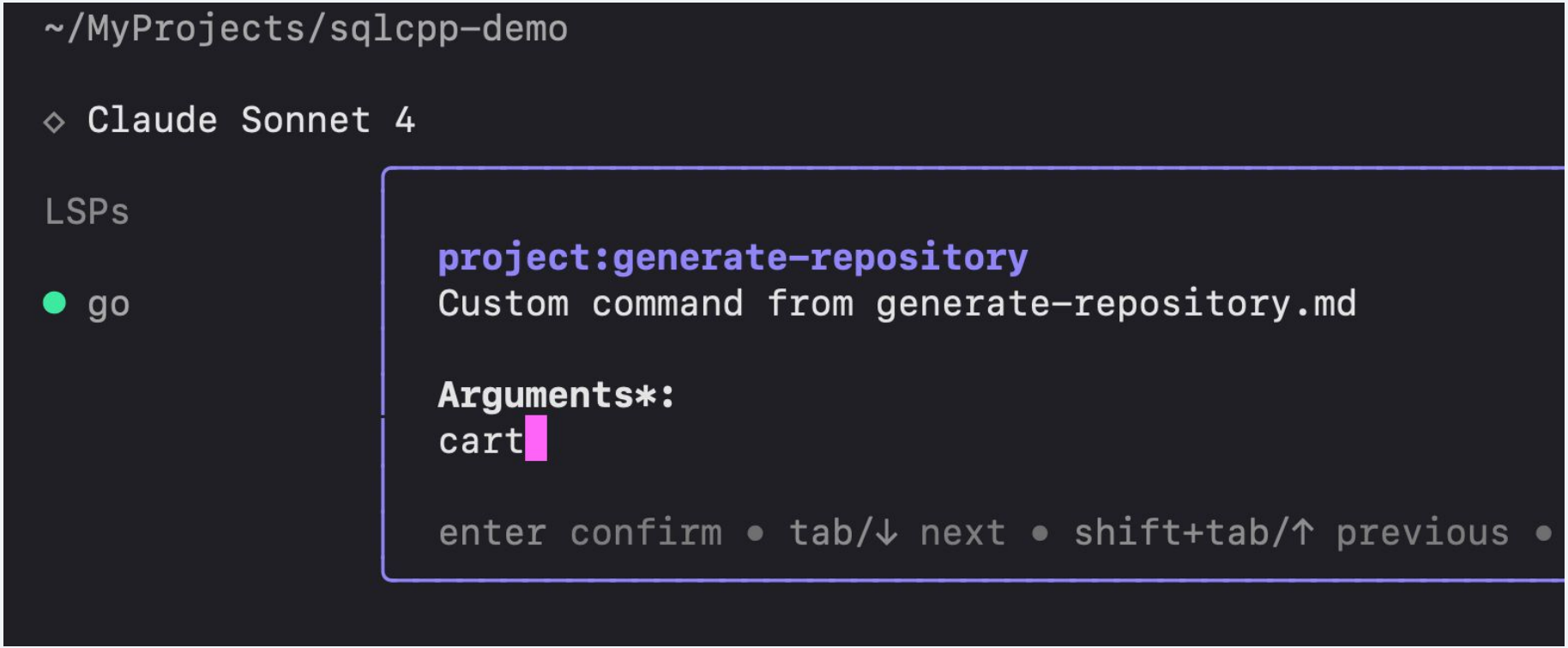
Reference implementation project

- 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
● go
MCPs
● 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
◇ Claude Sonnet 4
LSPs
● go
Commands
○ System ◉ User
> Type to filter
project:generate-repository
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

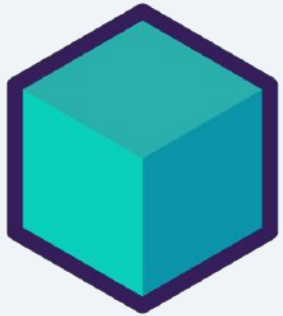
LSPs

● go ∵ 4

MCPs

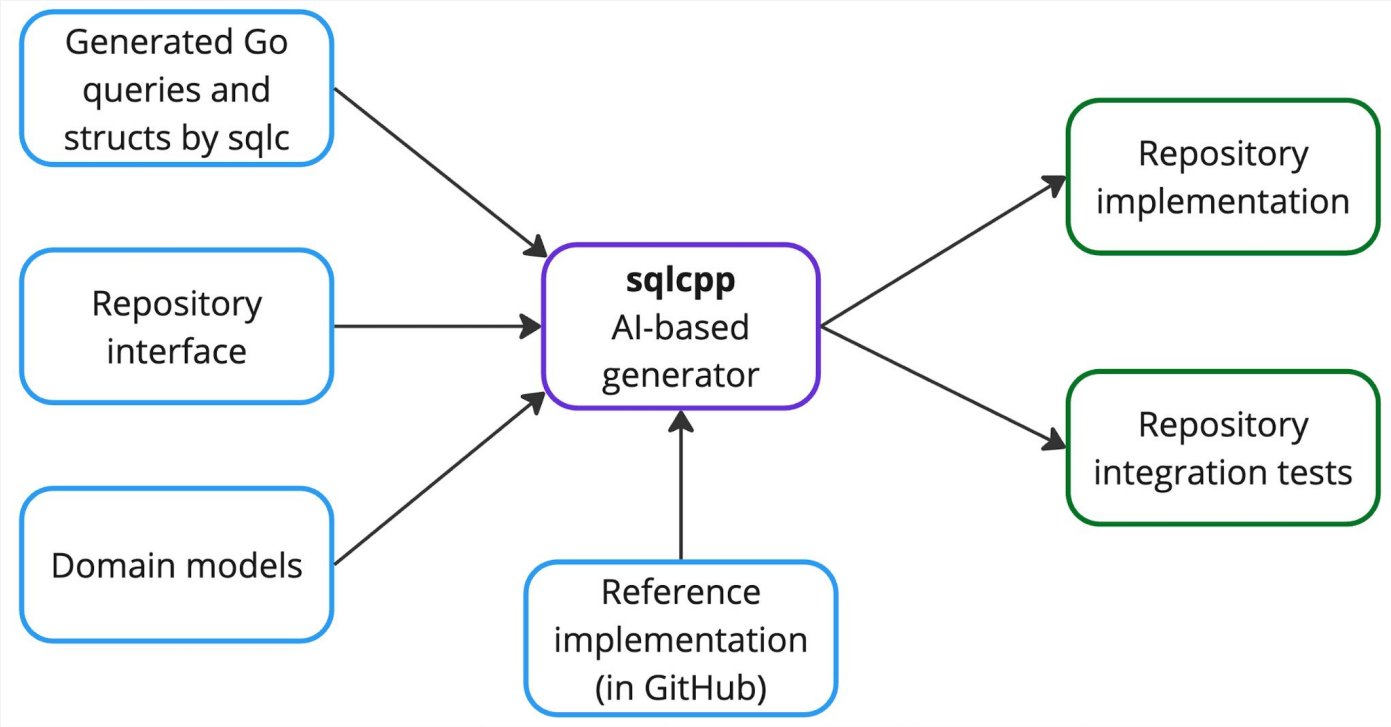
● godoc 1 tool

Testing repositories



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

sqlc++ architecture recap



Takeaways

- Adopt ***sqlc*** (and ***Crush***), contribute
- Avoid *sqlc*-generated structs in business logic
- Do not ignore transactions
- Repository boilerplate can be automated

Thank you!

Q&A

Nikolay Kuznetsov

nikolayk812

nkuznetsov

github.com/nikolayk812/sqlcpp