Highly Cohesive, Loosely Coupled (& Very Awesome)

A domain-driven approach to beautiful systems

Let's imagine tonight

We work at a hot new startup: Delorean. The Uber for time-travel!

OMG it's making so much money!

 We're releasing features right and left!

But the code is a mess!

As systems grow, they naturally want to fall into disarray...

The evolution of a feature

Feature: *As a passenger, I want to hail a Delorean*

- So I can travel in time!

```ruby
class TripsController
  def create
    passenger = Passenger.find(params[:passenger_id])
    driver = Driver.where(
      available: true,
      longitude: params[:longitude],
      latitude: params[:latitude]
    ).first
    driver.send_to!(passenger)
  end
end
```

The evolution of a feature

Feature: *As a passenger, I want to hail a Delorean*

- So I can travel in time!
- ...and my credit card will be charged

```ruby
class TripsController
  def create
    passenger = Passenger.find(params[:passenger_id])
    driver = Driver.where(
      available: true,
      longitude: params[:longitude],
      latitude: params[:latitude]
    ).first
    BraintreeService.charge(passenger)
    driver.send_to!(passenger)
  end
end
```

The evolution of a feature

Feature: *As a passenger, I want to hail a Delorean*

- So I can travel in time!
- ...and my credit card will be charged
- ...and the system should log the event to Google Analytics

```ruby
class TripsController
  def create
    passenger = Passenger.find(params[:passenger_id])
    driver = Driver.where(
      available: true,
      longitude: params[:longitude],
      latitude: params[:latitude]
    ).first
    BraintreeService.charge(passenger)
    AnalyticsService.log_ride_created!
    driver.send_to!(passenger)
  end
end
```

The evolution of a feature

Feature: *As a passenger, I want to hail a Delorean*

- So I can travel in time!
- ...and my credit card will be charged
- ...and the system should log the event to Google Analytics
- ...and we should totally also do food delivery

```ruby
class TripsController
  def create
    passenger = Passenger.find(params[:passenger_id])
    is_food = params[:ride_type] == 'food'
    driver = Driver.where(
      can_food_delivery: is_food,
      # ...
    ).first
    restaurant = Restaurant.find_by(meal_type: params[:m
    BraintreeService.charge(passenger)
    AnalyticsService.log_ride_created!
    driver.itinerary.add(restaurant)
    driver.itinerary.add(passenger)
  end
end
```

Code clutter in Rails

As the monolith grows, feature code is scattered across the app.

```
app/
  controllers/trips_controller.rb
  models/trip.rb
  helpers/trip_helper.rb
  services/calculate_trip_cost.rb
```

Code clutter in Rails

As the monolith grows, feature code is scattered across the app.

```
app/
  controllers/restaurants_controller.rb
  models/restaurant.rb
  models/meal.rb
  helpers/restaurant_helper.rb
  services/calculate_meal_cost.rb
```

Code clutter in Rails

As the monolith grows, feature code is scattered across the app.

```
app/
  controllers/puppy_deliveries_controller.rb
  models/puppy_delivery.rb
  models/animal_shelter.rb
```

So here we are...

- The code is tangled & difficult to change
- Regressions are common
- Features take forever to build & release

Hi, I'm Andrew

Friendly neighborhood programmer at Carbon Five

Carbon Five

Beautiful systems

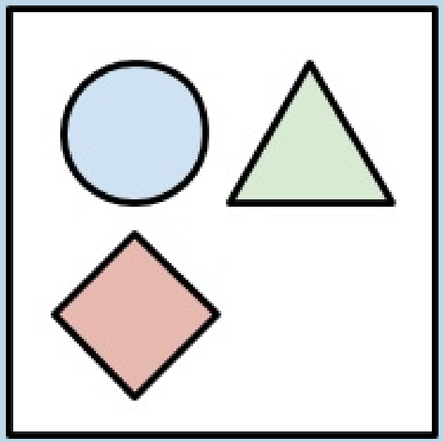
Beautiful systems are highly cohesive

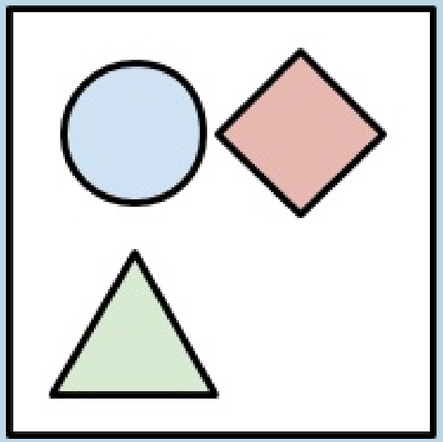
Beautiful systems are loosely coupled

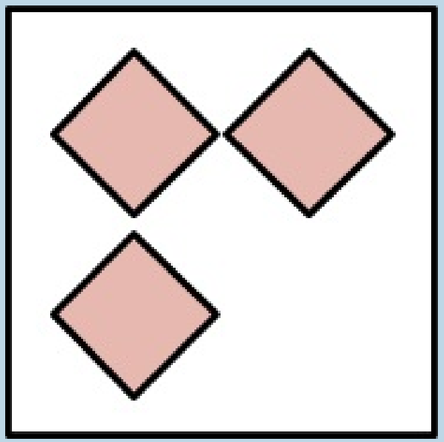
Highly cohesive

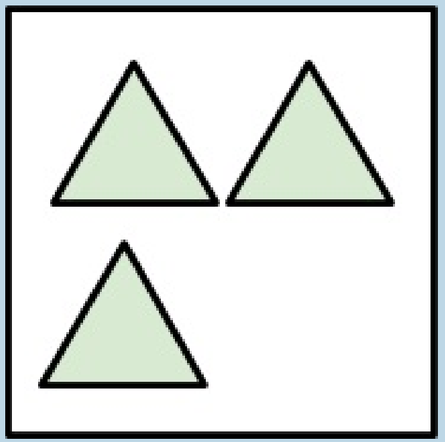
Elements of a module are strongly related to each other

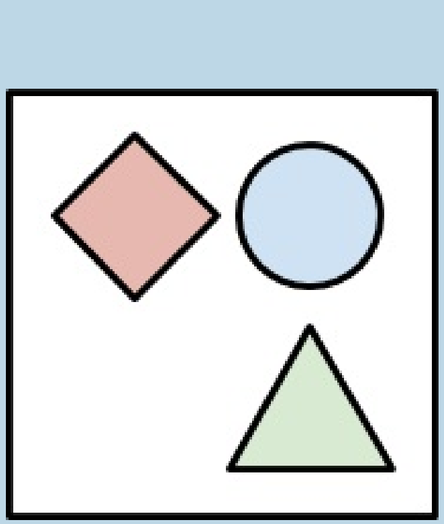
Near each other, are easily accessible.

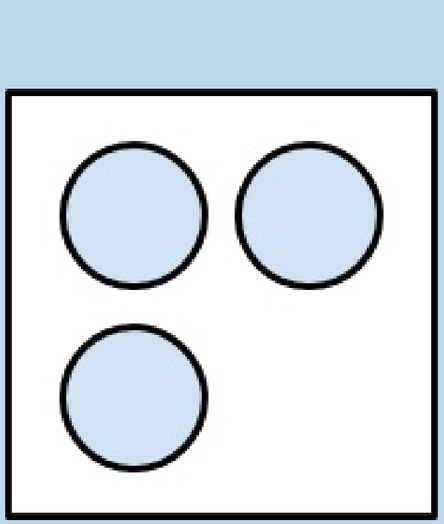












Low Cohesion

High Cohesion

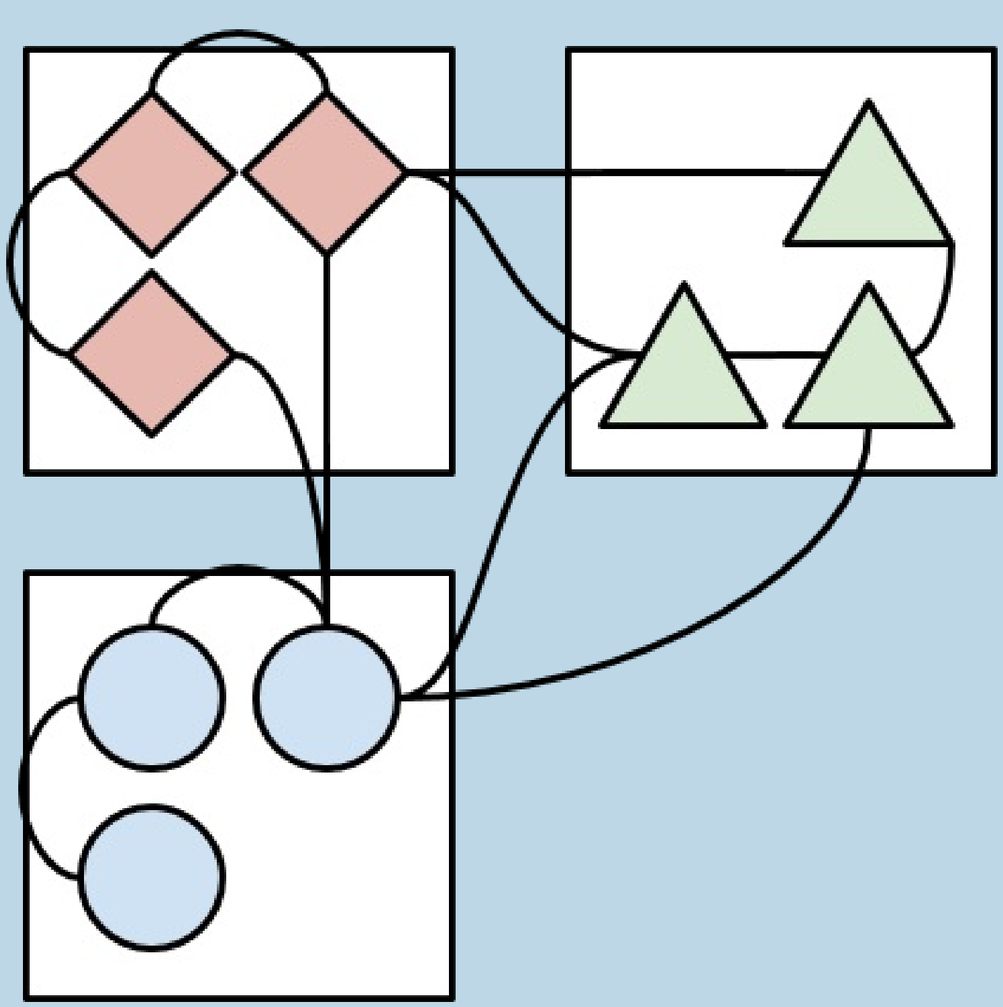
Loosely coupled

Modules minimize their dependencies so that they are easily modifiable

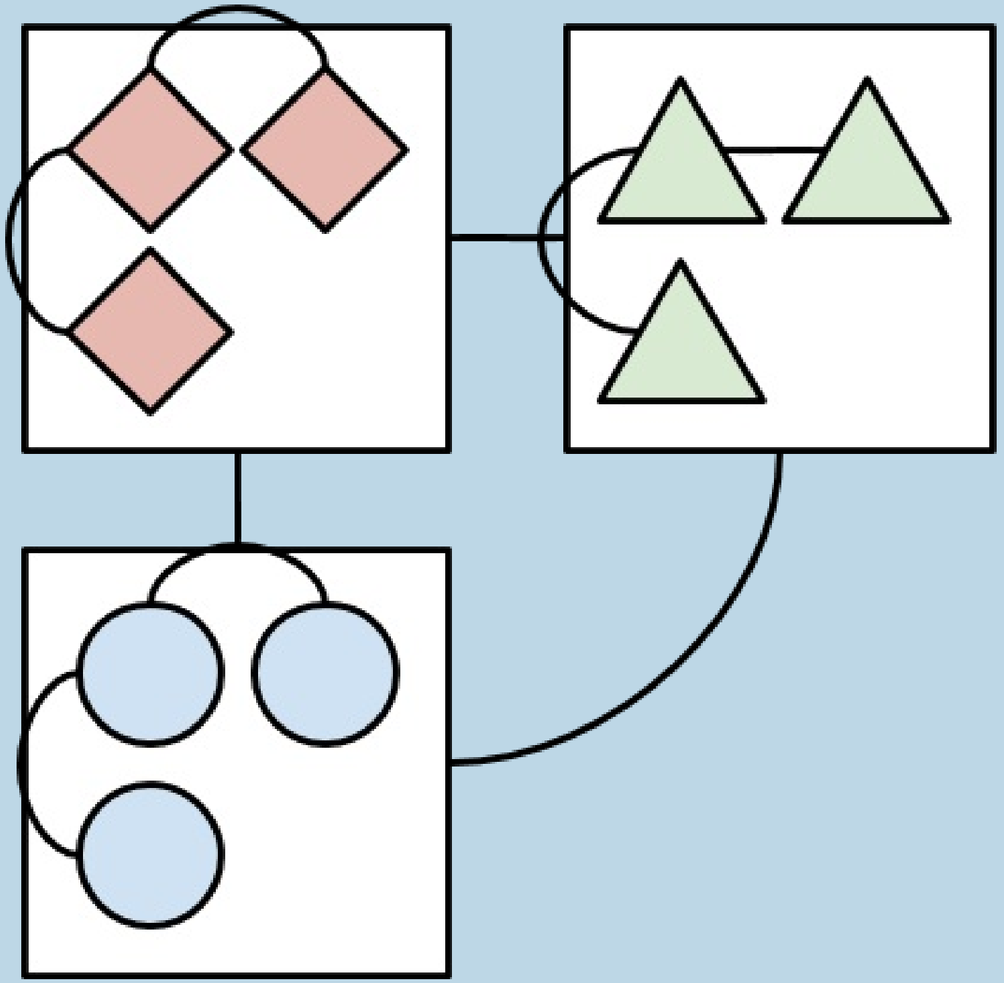
Loosely coupled

Modules minimize their dependencies so that they are easily modifiable

Can evolve independently of other modules in the system



Tight Coupling



Loose Coupling

Introducing Domain-Driven Design

DDD is both a set of high-level design activities and specific software patterns

Our goals tonight

- **Visualize our system** from a domain perspective.

Our goals tonight

- **Visualize our system** from a domain perspective.

- ✏ Learn insights to **draw boundaries** in our code!

Our goals tonight

- **Visualize our system** from a domain perspective.

- ✏ Learn insights to **draw boundaries** in our code!

- Do a little bit of **refactoring**.

Strategic Design

Through an exercise called Context Mapping

Apply It! ⚡

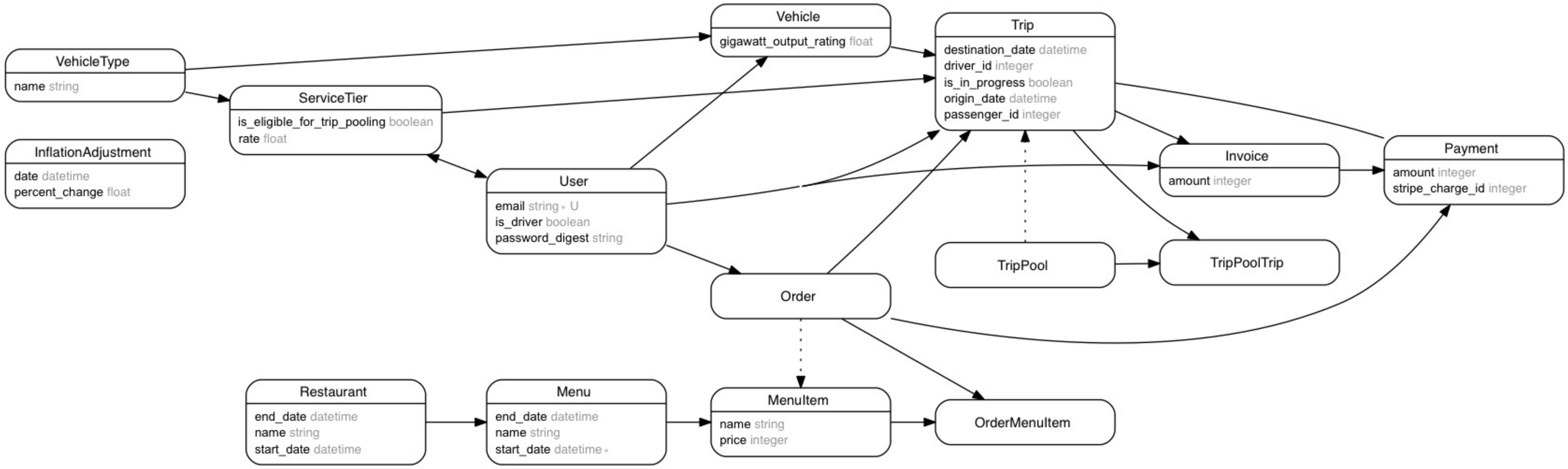
Step 1: Visualize Your System

Let's generate an ERD diagram!

I like to generate mine with a gem like `railroady` or `rails-erd`

If you have multiple systems, do this for each system.

Delorean domain model



Yikes.

Core domain

The **Core Domain** is the thing that your business does that makes it unique.

Core domain

The **Core Domain** is the thing that your business does that makes it unique.

Delorean Core Domain: **Transportation**

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

Delorean Supporting Domains:

- **Driver Routing** (route me from X to Y)

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

Delorean Supporting Domains:

- **Driver Routing** (route me from X to Y)
- **Notifications** (push notifications)

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

Delorean Supporting Domains:

- **Driver Routing** (route me from X to Y)
- **Notifications** (push notifications)
- **Financial Transactions** (charge the card)

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

Delorean Supporting Domains:

- **Driver Routing** (route me from X to Y)
- **Notifications** (push notifications)
- **Financial Transactions** (charge the card)
- **Product Analytics** (track business metrics)

Supporting domains

A **Supporting Domain** (or Subdomain) are the areas of the business that play roles in making the **Core Domain** happen.

Delorean Supporting Domains:

- **Driver Routing** (route me from X to Y)
- **Notifications** (push notifications)
- **Financial Transactions** (charge the card)
- **Product Analytics** (track business metrics)
- **Customer Support** (keep people happy)

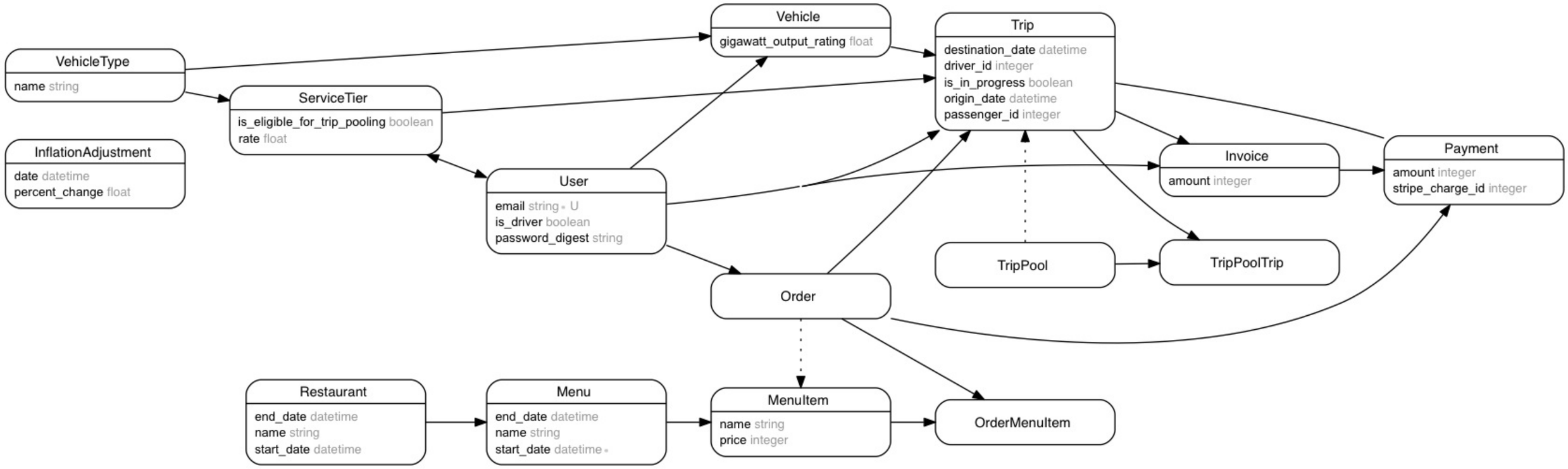
Apply It! ⚡

Step 2: Draw domains on your diagram

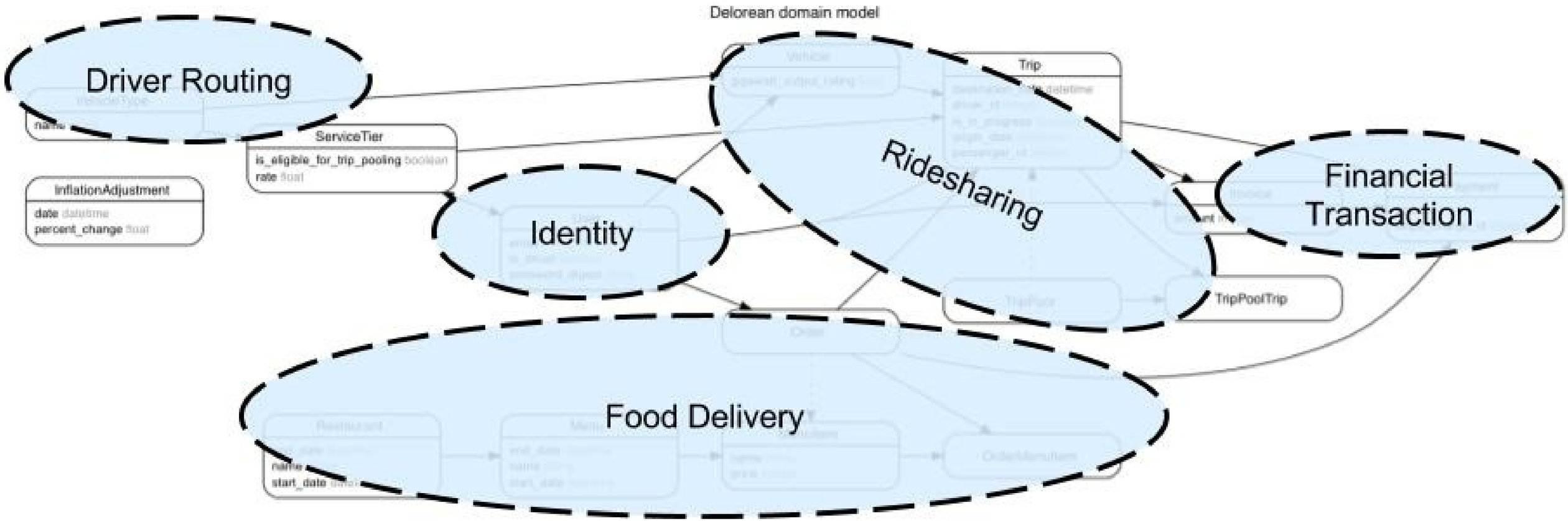
Overlay your domains on top of the ERD diagram

You might discover some domains you never even thought you had!

Delorean domain model



Delorean domain model



Bounded Contexts

A **Bounded Context** is:

- Concretely: a software system (like a codebase)
- Linguistically: a delineation in your domain where concepts are "bounded", or contained

Apply It! ⚡

Step 3: Overlay your bounded contexts

Next up - with a different color pen or marker, draw lines around system boundaries / bounded contexts.

Apply It! ⚡

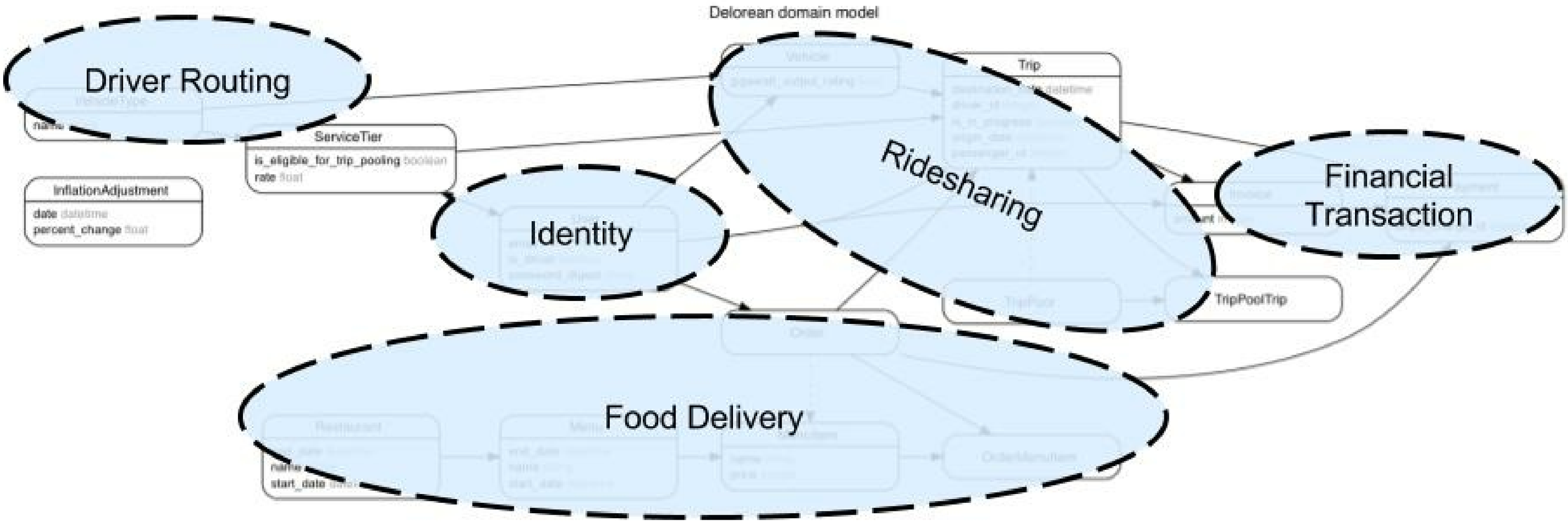
Step 3: Overlay your bounded contexts

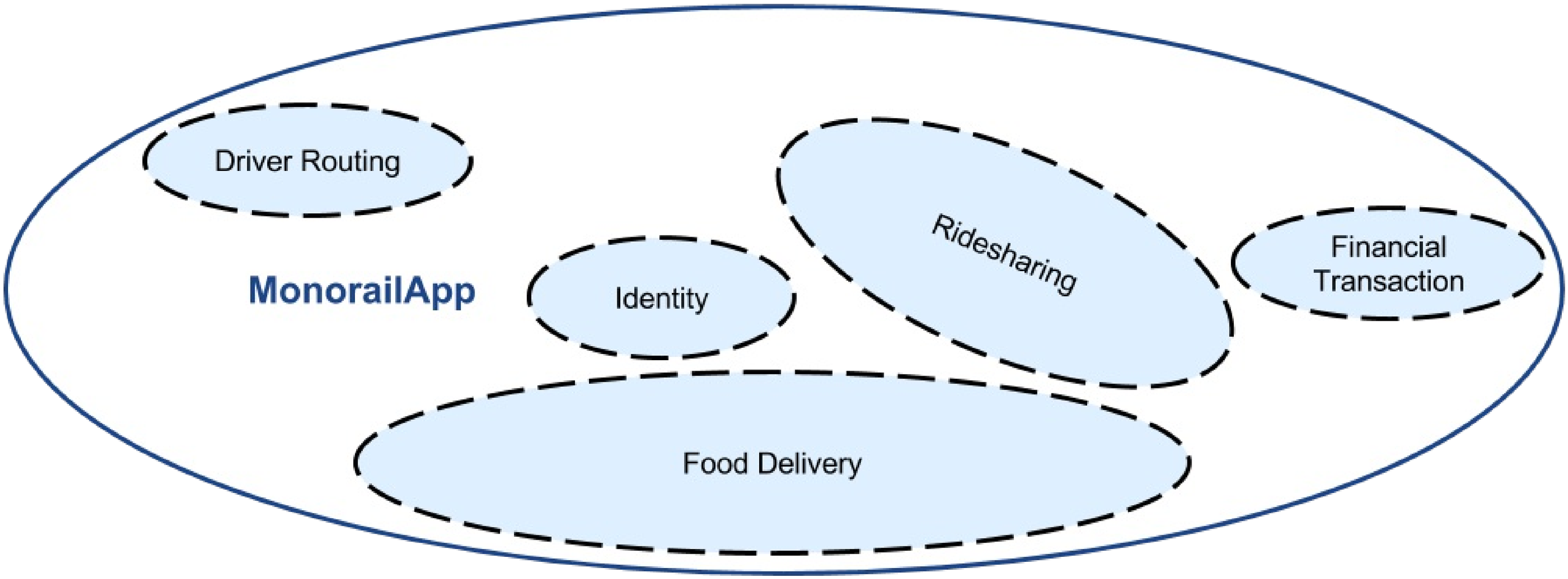
Next up - with a different color pen or marker, draw lines around system boundaries / bounded contexts.

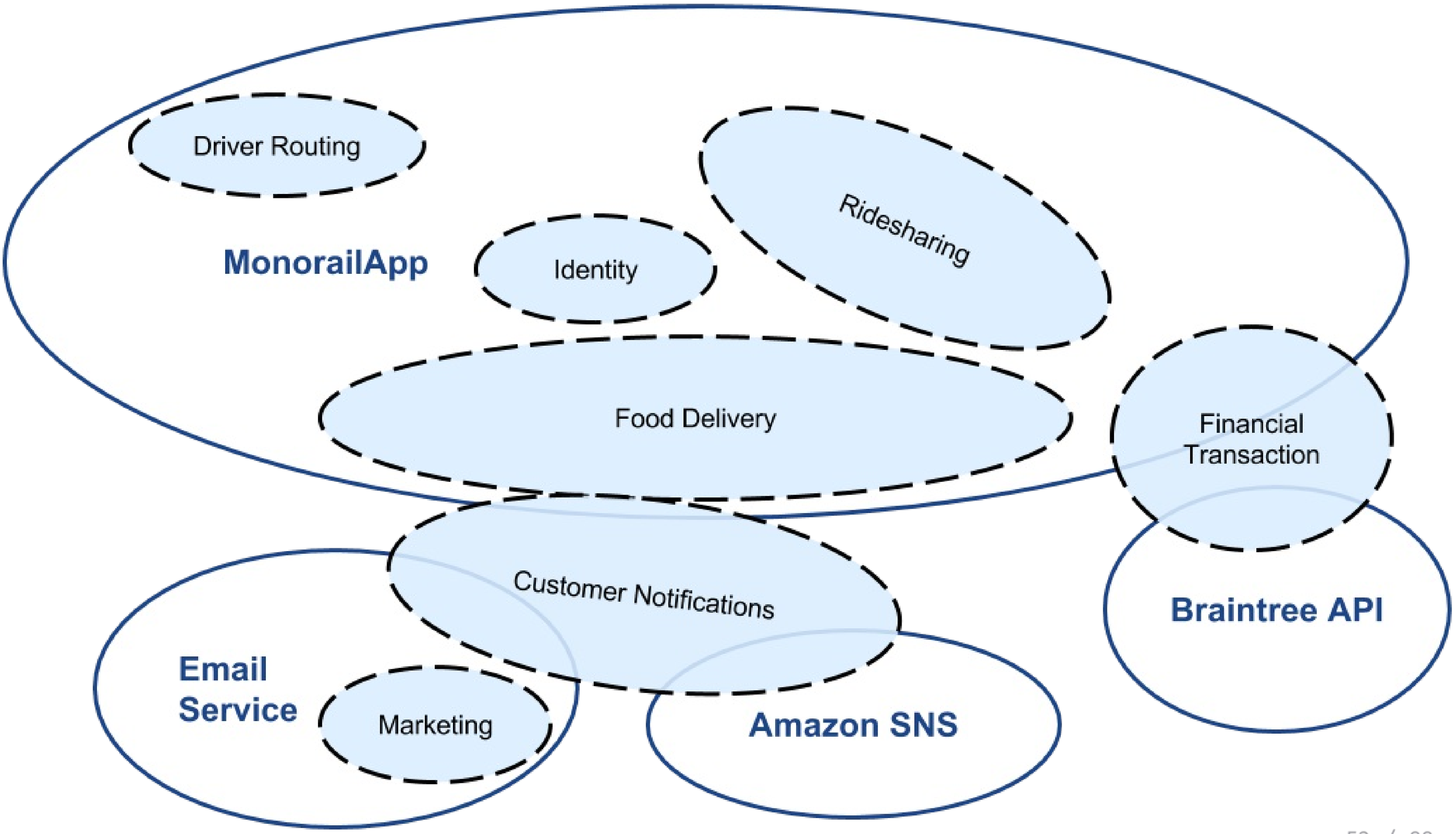
You may also find other system boundaries like:

- External cloud providers
- Other teams' services or systems

Delorean domain model




Driver Routing
MonorailApp
Identity
Ridesharing
Financial Transaction
Food Delivery


Driver Routing
MonorailApp
Identity
Ridesharing
Food Delivery
Financial Transaction
Customer Notifications
Email Service
Marketing
Amazon SNS
Braintree API

Congrats! You just made a Context Map!

A **Context Map** gives us a place to see the current system as-is (the problem space), the strategic domains, and their dependencies.

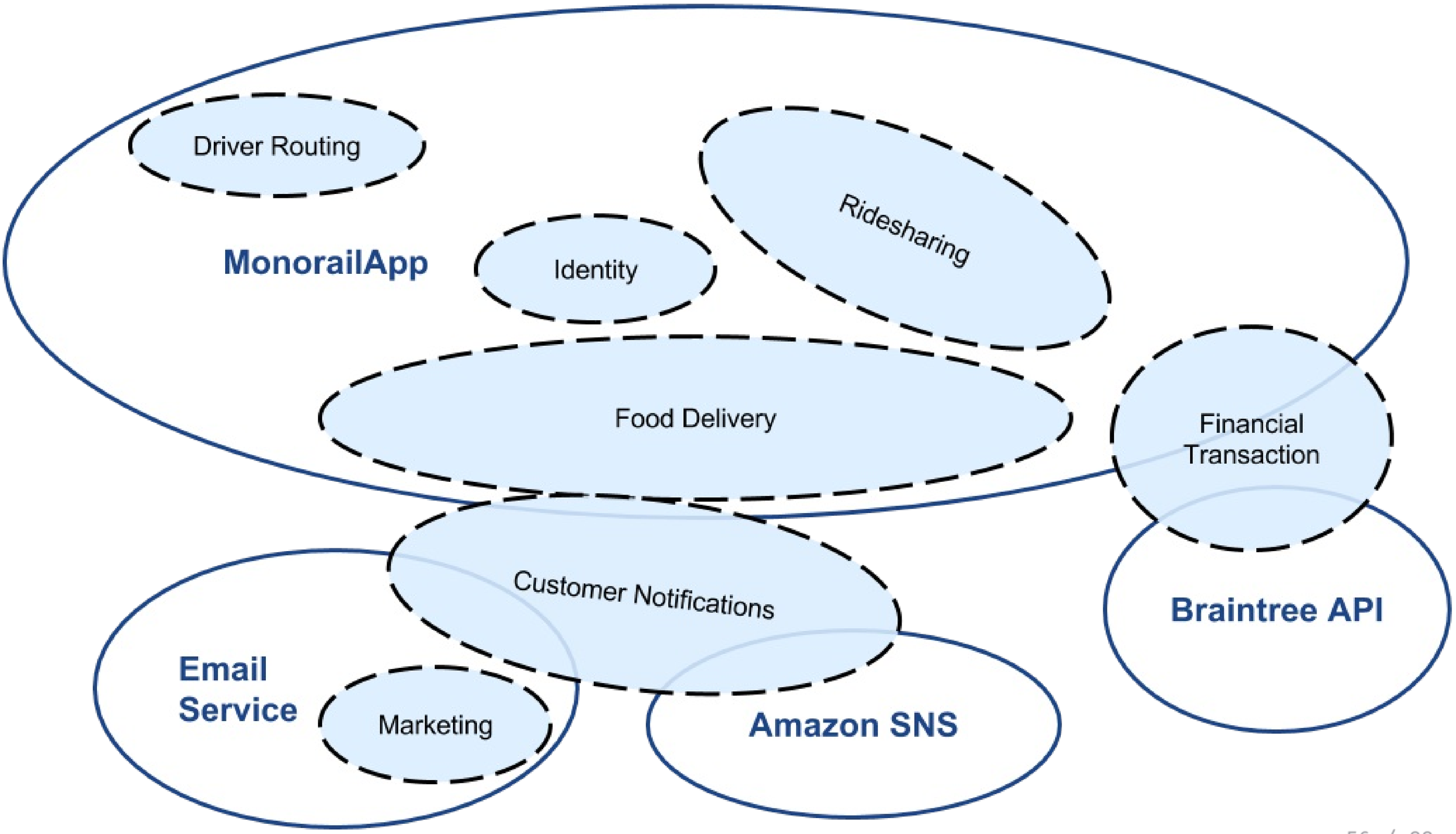
Making sense of the Context Map

We may notice a few things:

Making sense of the Context Map

We may notice a few things:

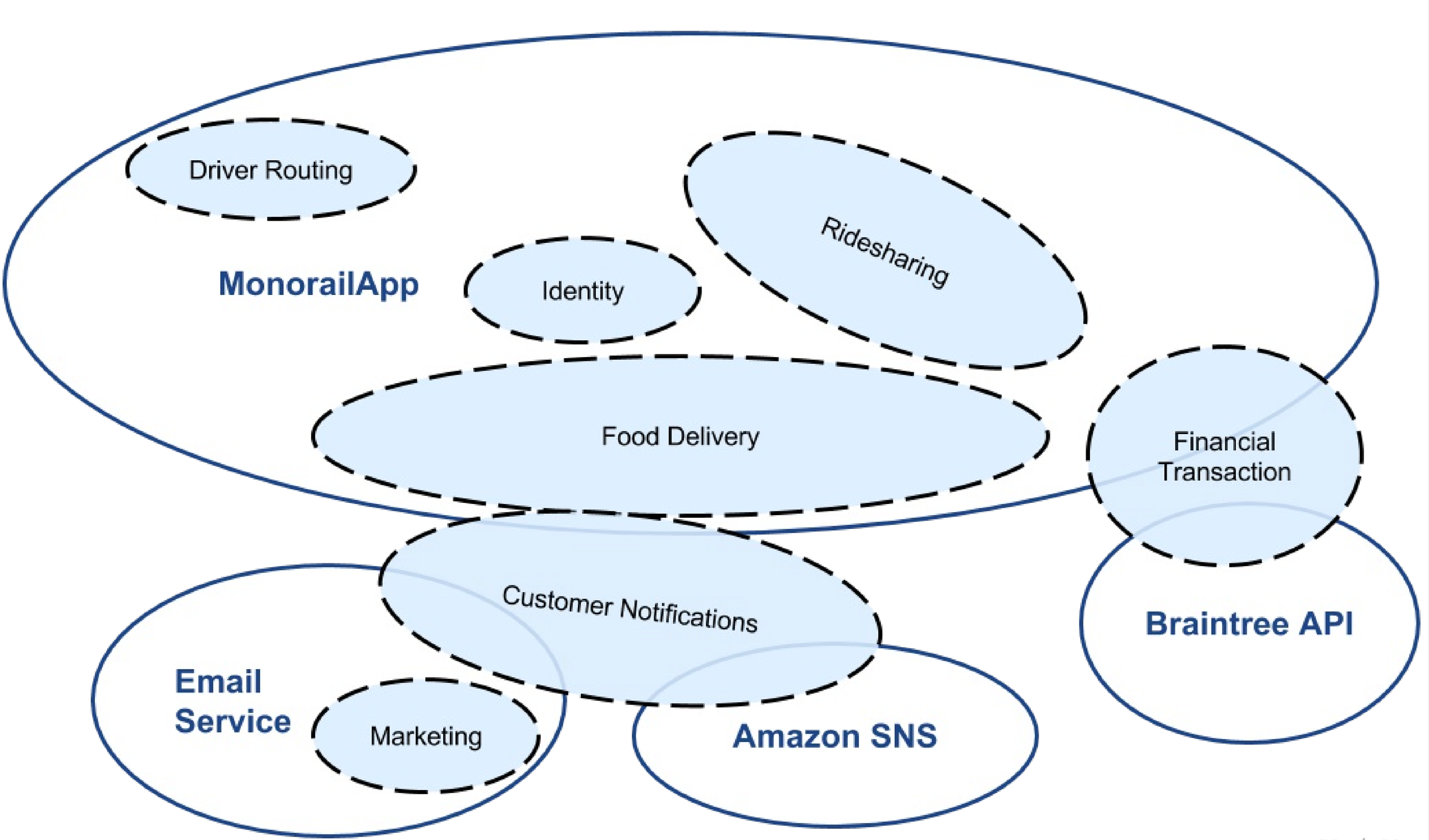
- One bounded context contains multiple sub-(supporting) domains


Driver Routing
MonorailApp
Identity
Ridesharing
Food Delivery
Financial Transaction
Customer Notifications
Email Service
Marketing
Amazon SNS
Braintree API

Making sense of the Context Map

We may notice a few things:

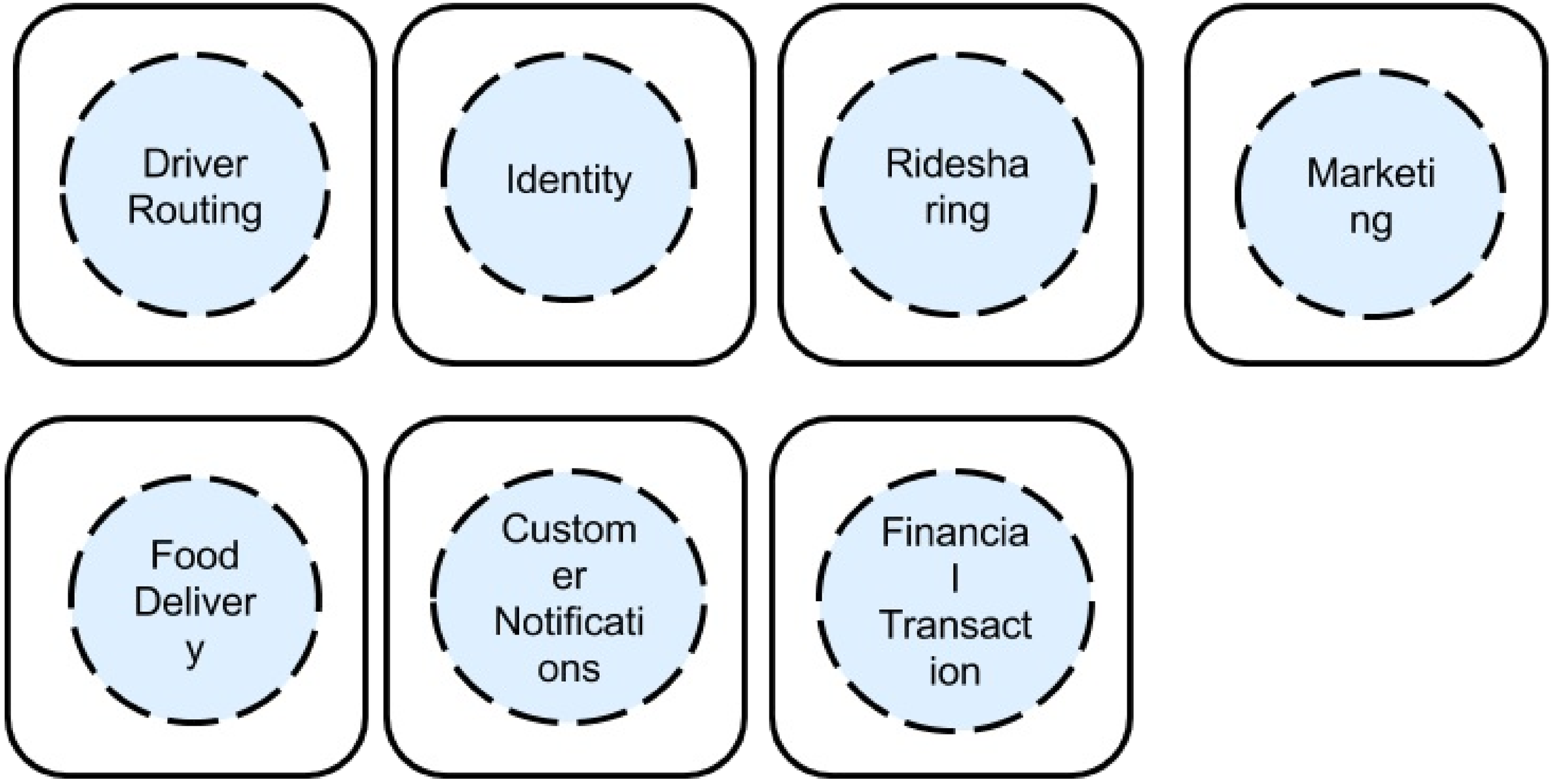
- One bounded context contains multiple sub-(supporting) domains
- Multiple bounded contexts are required to support a single domain


Driver Routing
MonorailApp
Identity
Ridesharing
Food Delivery
Financial Transaction
Customer Notifications
Email Service
Marketing
Amazon SNS
Braintree API

An Ideal Architecture

Each **Domain** should have its own **Bounded Context**

Key concept in DDD!


Driver Routing
Identity
Ridesha ring
Marketi ng
Food Deliver y
Custom er Notificati ons
Financia l Transact ion

Increased cohesion!

We just found the areas where code "naturally" fits together, because they are serving the same business goal.

Apply It! ⚡

Break your application into domain modules

Incremental refactoring, using Ruby Modules to lead the way!

```ruby
class Trip < ActiveRecord::Base
  belongs_to :vehicle
  belongs_to :passenger
  belongs_to :driver
end

class TripsController < ApplicationController
  # ...
end
```

```ruby
module Ridesharing
  class Trip < ActiveRecord::Base
    belongs_to :vehicle
    belongs_to :passenger
    belongs_to :driver
  end
end

module Ridesharing
  class TripsController < ApplicationController
    # ...
  end
end
```

Find references to newly modulized classes and change them.

```ruby
# config/routes.rb
resources :trips
```

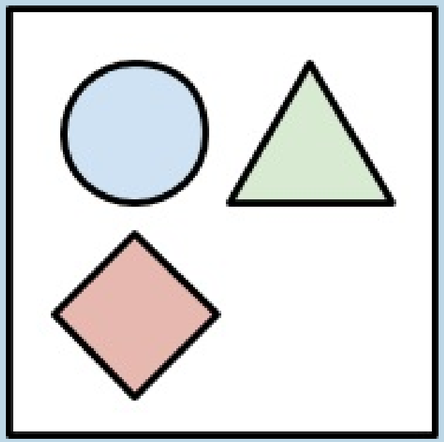
```ruby
# config/routes.rb

namespace :ridesharing, path: '/' do
  resources :trips
end
```

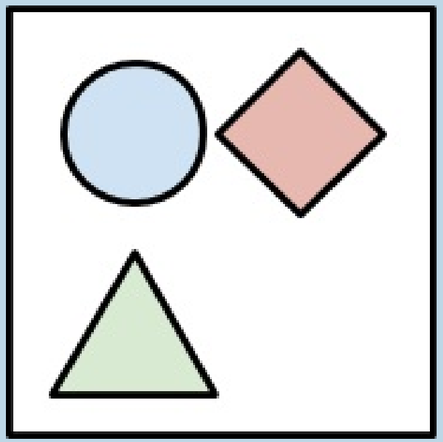
```ruby
class Invoice
  belongs_to :trip
end
```

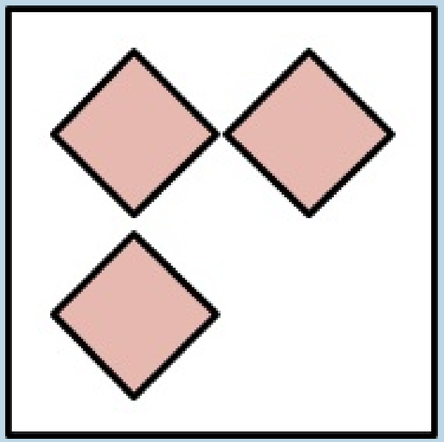
```ruby
class Invoice
  belongs_to :trip, class_name: Ridesharing::Trip
end
```

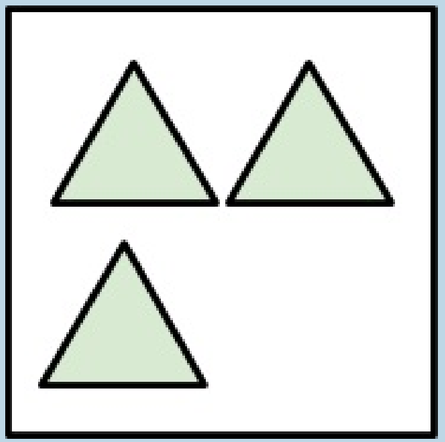
Creating domain-oriented folders

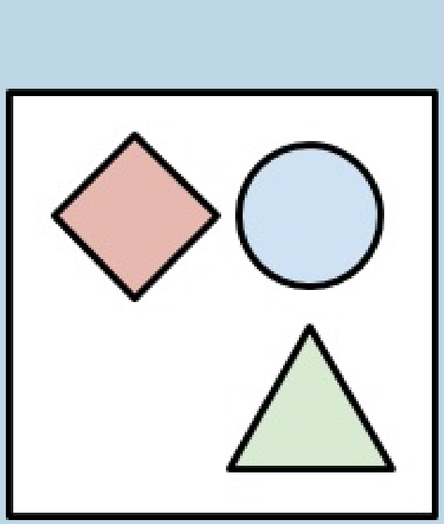
```
app/domains/ridesharing/trip.rb
app/domains/ridesharing/service_tier.rb
app/domains/ridesharing/vehicle.rb
app/domains/ridesharing/trips_controller.rb
app/domains/ridesharing/trips/show.html.erb
```

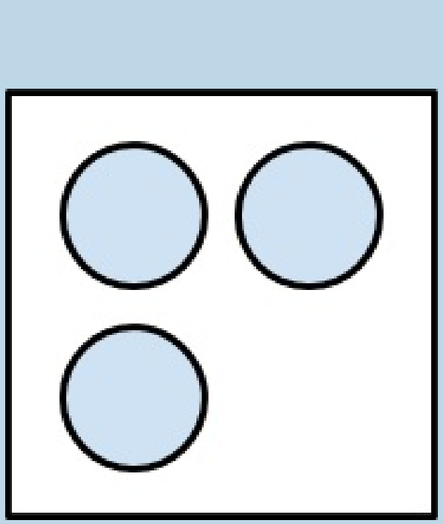












Low Cohesion

High Cohesion

Hence: modulizing increases cohesion

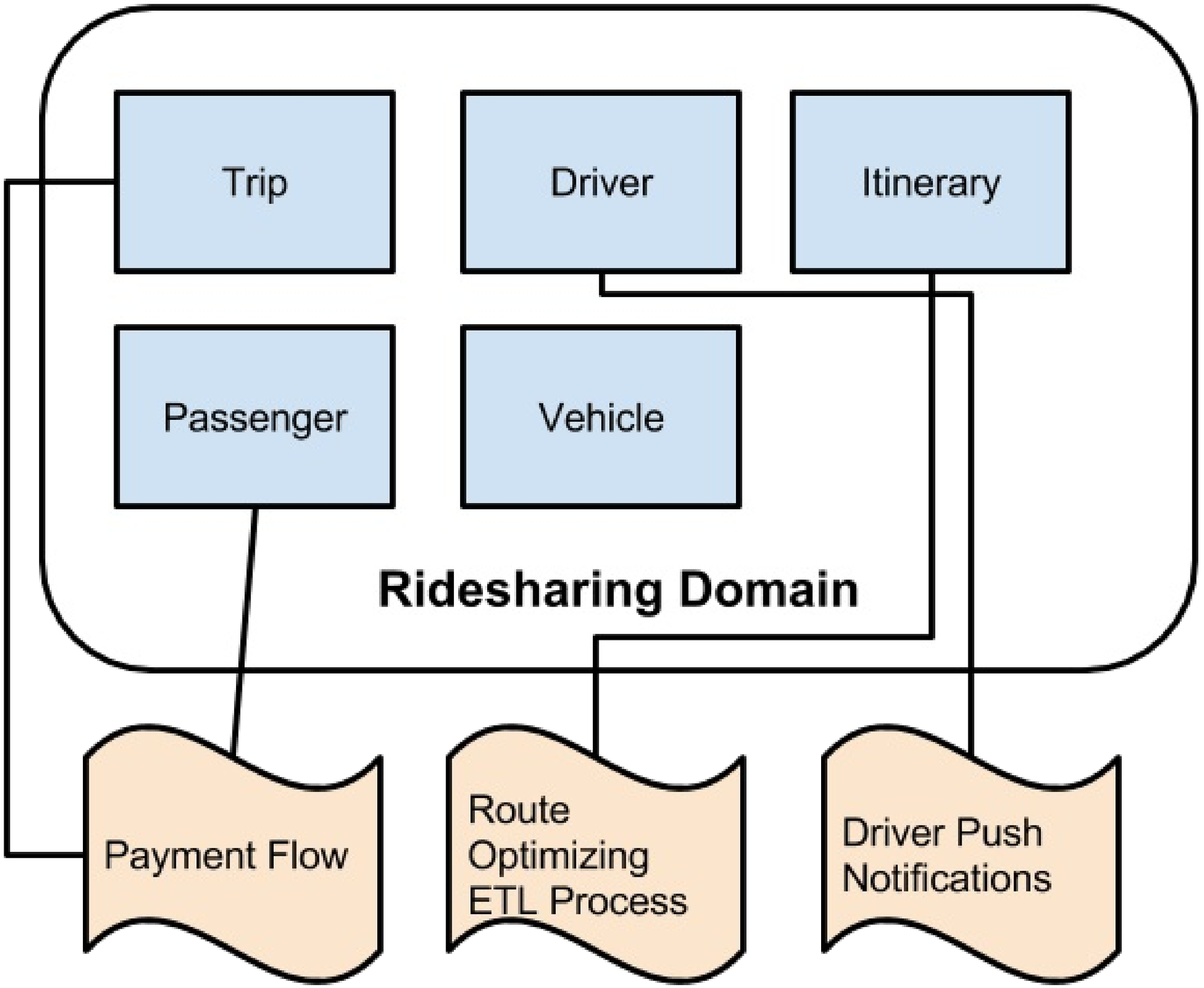
Let's move on to coupling...

ActiveRecord relationships can be abused!

Objects start knowing too much about the entire world.

"God Objects"

```ruby
class PaymentConfirmation
  belongs_to :trip, class_name: Ridesharing::Trip
  belongs_to :passenger, class_name: Ridesharing::Passen
  belongs_to :credit_card
  has_many :menu_items
  belongs_to :coupon_code
  has_one :retriable_email_job
  # ad infinitum...
end
```


Trip
Driver
Itinerary
Passenger
Vehicle
Ridesharing Domain
Payment Flow
Route
Optimizing
ETL Process
Driver Push
Notifications

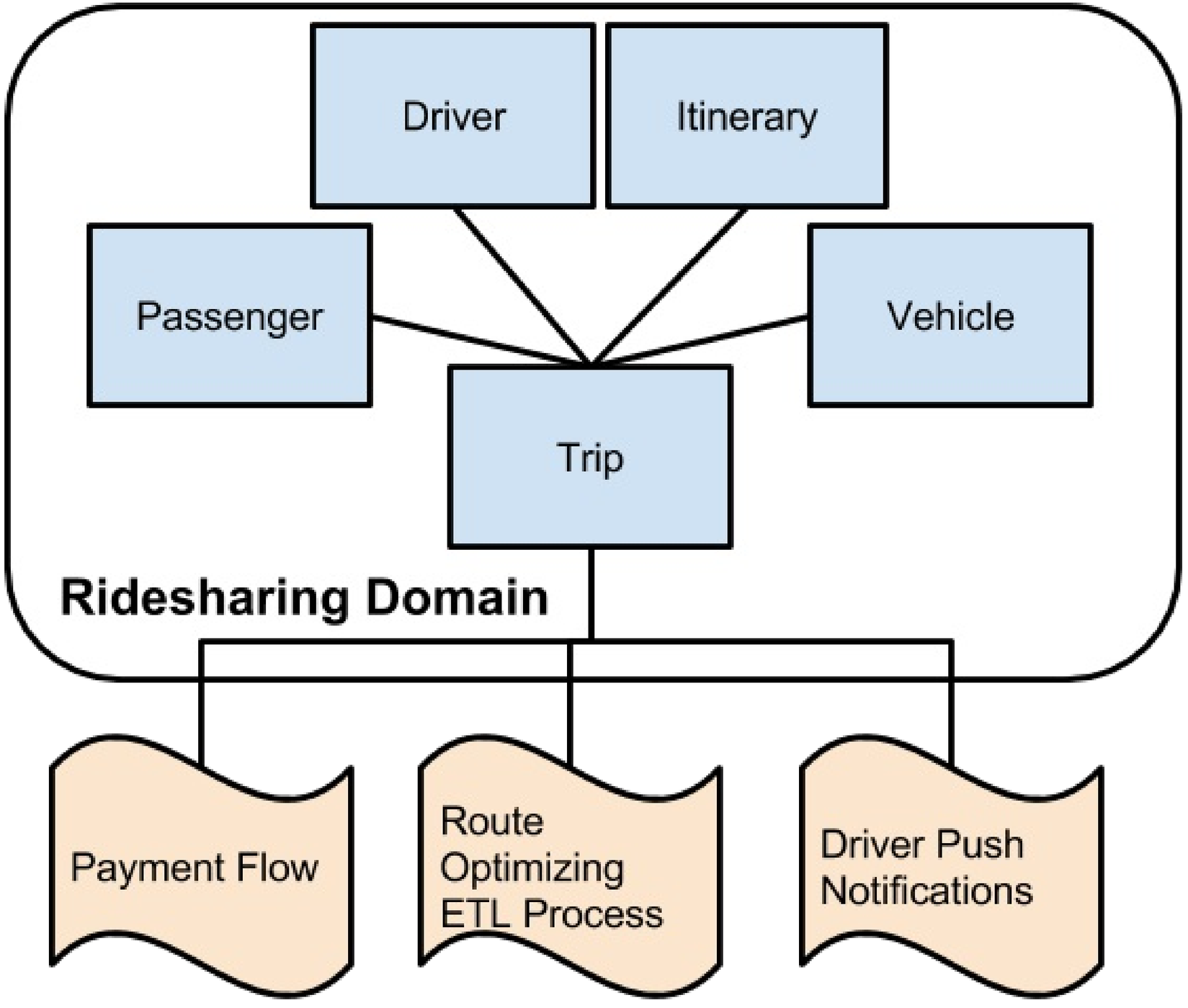
Aggregate Roots

Aggregate Roots are top-level domain models that reveal an object graph of related entities beneath them.

Aggregate Roots

Aggregate Roots are top-level domain models that reveal an object graph of related entities beneath them.

Can be considered a **Facade**


Driver
Itinerary
Passenger
Vehicle
Trip
Ridesharing Domain
Payment Flow
Route
Optimizing
ETL Process
Driver Push
Notifications

Decrease coupling by only exposing aggregate roots

Make it a rule in your system that you may only access another domain's **Aggregate Root**.

Decrease coupling by only exposing aggregate roots

Make it a rule in your system that you may only access another domain's **Aggregate Root**.

Internally, it's OK to reach for whatever you need.

Make service objects that provide Aggregate Roots

Your source domain can provide a service (Adapter) that returns the **Aggregate Root**

```ruby
module Ridesharing
  class FetchTrip
    def call(id)
      Trip
        .includes(:passenger,
                  :trip, ...)
        .find(id)
      # Alternatively, return something custom
      # OpenStruct.new(trip: Trip.find(id), ...)
    end
  end
end
```

```ruby
class PaymentConfirmation
  belongs_to :trip, class_name: Ridesharing::Trip
  belongs_to :passenger, class_name: Ridesharing::Passen
  # ...
end
```

```ruby
class PaymentConfirmation
  def trip
    # Returns the Trip aggregate root
    Ridesharing::FetchTrip.new.find(payment_id)
  end
end

# OLD: payment_confirmation.passenger
# NEW: payment_confirmation.trip.passenger
```

Decrease coupling by publishing events for async dependencies

Domains that only need unidirectional data flow work well here!

Just send an event notifying the outside world!

Instead of needing to know about the outside world, we simply publish an event.

```ruby
# Old way
class TripController
  def create
    # ...
    ReallySpecificGoogleAnalyticsThing
      .tag_manager_logging('trip_created',
                           ENV['GA_ID'],
                           trip)
  end
end
```

```ruby
class TripController
  def create
    # ...
    EventPublisher.publish(:trip_created, trip.id)
  end
end
```

Conway's Law and DDD

Conway's Law, paraphrased: "Software systems tend to look like the organizations that produce them"

Conway's Law and DDD

Conway's Law, paraphrased: "Software systems tend to look like the organizations that produce them"

DDD modeling oftentimes reveals domains that follow organizational layouts.

Your software systems follow organizational optimizations.

Conway's Law and DDD

Conway's Law, paraphrased: "Software systems tend to look like the organizations that produce them"

DDD modeling oftentimes reveals domains that follow organizational layouts.

Your software systems follow organizational optimizations.

Thus this is a very natural place to draw a seam!

Warning: Limitations apply!

Don't try to do this on every project!

Warning: Limitations apply!

Don't try to do this on every project!

I've been guilty of overdesigning.

Warning: Limitations apply!

Don't try to do this on every project!

I've been guilty of overdesigning.

Try it out, step by step

Warning: Limitations apply!

Don't try to do this on every project!

I've been guilty of overdesigning.

Try it out, step by step

Back it out if this doesn't "fit"

What we did tonight

NO GLYPH **Visualized our system** with a Context Map

✏ **Drew boundaries** in our code!

NO GLYPH Do a little bit of **refactoring** with domain modules, Aggregate Roots and Domain Events.

Sample code:

https://www.github.com/andrewhao/delorean

Thanks!

Github: andrewhao

Twitter: @andrewhao

Email: andrew@carbonfive.com

Prior Art

- W. P. Stevens ; G. J. Myers ; L. L. Constantine. "Structured Design" - IBM Systems Journal, Vol 13 Issue 2, 1974
- Evans, Eric. Domain Driven Design
- Vernon, Vaughan. Implementing Domain-Driven Design
- http://www.win.tue.nl/~wstomv/quotes/structured-design.html#6
- https://www.infoq.com/articles/ddd-contextmapping
- http://gorodinski.com/blog/2013/04/29/sub-domains-and-bounded-contexts-in-domain-driven-design-ddd/