

Taming the Beast

Safely Managing Database Operations in Rails
in a Team of 100s

Who am I?

Who am I?

- Engineer on team-datastores at Intercom

Who am I?

- Engineer on team-datastores at Intercom
- Directly responsible for databases, caching, etc.

Who am I?

- Engineer on team-datastores at Intercom
- Directly responsible for databases, caching, etc.
- Hold (and raise!) the bar for productivity and availability

Who am I?

- Engineer on team-datastores at Intercom
- Directly responsible for databases, caching, etc.
- Hold (and raise!) the bar for productivity and availability
- Third time speaking at Rails World!

Quick disclaimer

Quick disclaimer

- Lots of information is going to be MySQL specific

Quick disclaimer

- Lots of information is going to be MySQL specific
- Concepts should generalise to any database tech

How do we think about productivity?

How do we think about productivity?

- Rails has great resources for learning

How do we think about productivity?

- Rails has great resources for learning
- Things that work when you're small don't necessarily work at scale

How do we think about productivity?

- Rails has great resources for learning
- Things that work when you're small don't necessarily work at scale
- Even so, we should try and ensure that the interface for engineers remains the same

How do we think about productivity?

- Rails has great resources for learning
- Things that work when you're small don't necessarily work at scale
- Even so, we should try and ensure that the interface for engineers remains the same
- Even more important now that LLMs are doing a lot of the driving!

You've just started a new Rails app

You've just started a new Rails app

- You probably don't have much if any real data in your database.

You've just started a new Rails app

- You probably don't have much if any real data in your database.
- You probably don't need zero downtime deployments.

You've just started a new Rails app

- You probably don't have much if any real data in your database.
- You probably don't need zero downtime deployments.
- Basically, you can get away with a lot.

Scale is a nice problem to have

Scale is a nice problem to have

- Issues will happen as you grow, it's inevitable

Scale is a nice problem to have

- Issues will happen as you grow, it's inevitable
- The important thing is to learn and not keep making the same mistakes

Scale is a nice problem to have

- Issues will happen as you grow, it's inevitable
- The important thing is to learn and not keep making the same mistakes
- Remember: checklists get written in blood

Checklists

Checklists

Checklists

- Document the edge cases you're aware of

Checklists

- Document the edge cases you're aware of
- Great tool for new people onboarding to understand the operational parameters

Checklists

- Document the edge cases you're aware of
- Great tool for new people onboarding to understand the operational parameters
- Start small and don't clutter them up!

Checklists

- Document the edge cases you're aware of
- Great tool for new people onboarding to understand the operational parameters
- Start small and don't clutter them up!
- Every checklist item is a liability

Checklists

Checklists

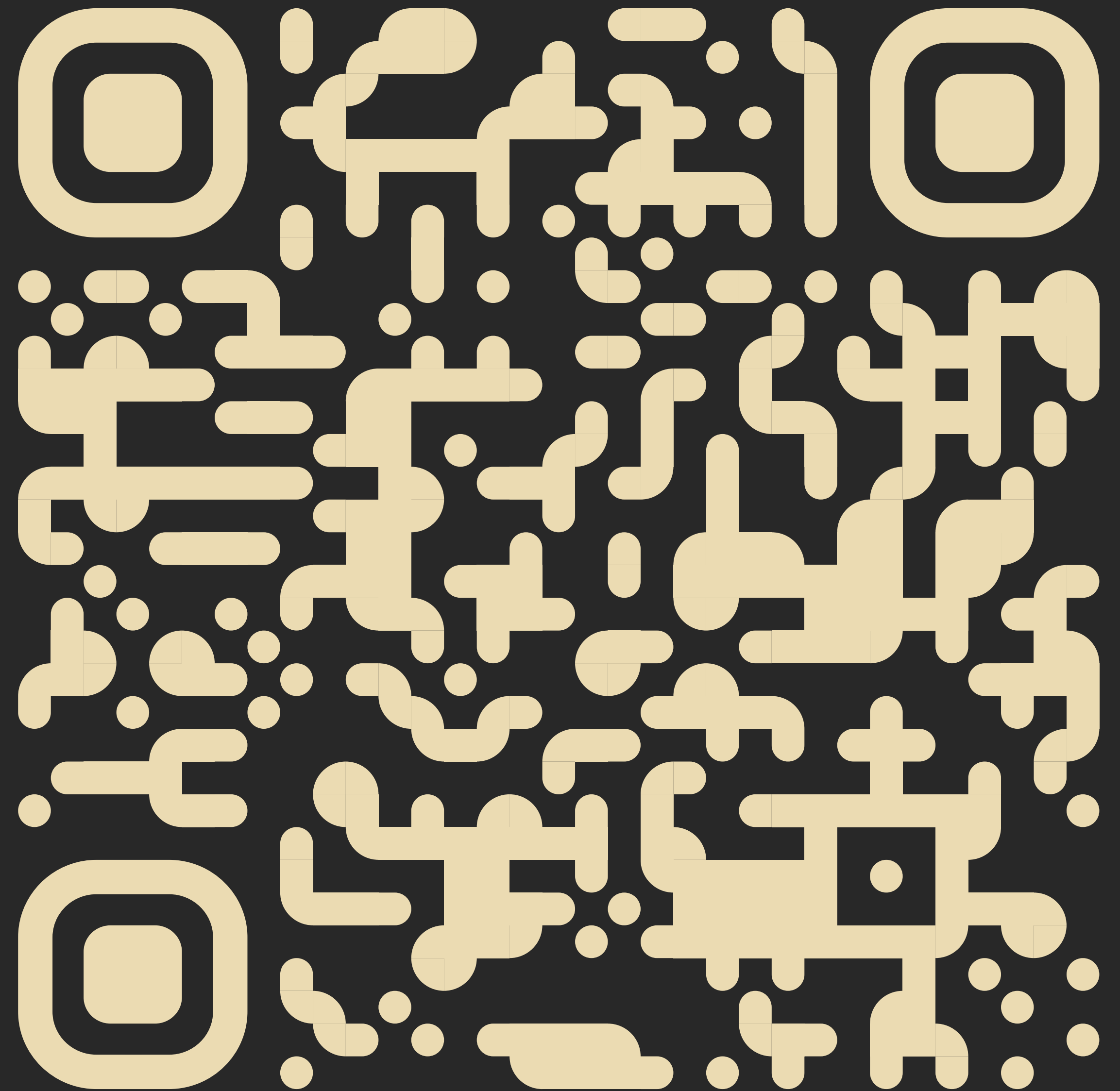
- Your checklist is only useful if everyone is using it

Checklists

- Your checklist is only useful if everyone is using it
- Meet people where they're at – if they're opening a migration PR they're in Github doing PR review.

Migration checklist poster

Github action to
automatically post a
checklist on migration
PRs



Default checklist

:paperclip: It looks like you are trying to merge a migration!

You should follow this checklist:

- [] Before removing columns: ensure the columns have been added to ``self.ignored_columns`` on the model in a previous pull request.
- [] Ensure proper indexes are in place if this changes a default scope.
- [] Ensure your migration PR contains only your migration files and updated schema files. No files outside the ``db`` directory should be changed.
- [] ****Don't merge the PR with your application changes before you complete all the steps in this checklist.****
- [] If you're making more than one change then use ``change_table`` and set ``bulk: true`` in the options. This will run multiple changes in a single statement so that if a single statement fails the server will rollback the entire operation, instead of leaving the migration in a half-done. If in doubt, use ``change_table :table_name, bulk: true do |t|`` everywhere.
- [] Your PR must contain the schema.rb file updated. If your migration file has an older timestamp, you leave (version: timestamp) untouched in schema.rb as you can't make the migration go back in time.
- [] Ensure you have run both ``db:migrate:up VERSION=YYYYMMDDHHMMSS`` and ``db:migrate:down VERSION=YYYYMMDDHHMMSS`` on your local machine, where the version timestamp corresponds to the timestamp in your generated migration filename.
- [] Add the output of the ``db:migrate:up`` and ``db:migrate:down`` commands as a comment to this PR.
- [] Merge your PR
- [] Wait until your change has been deployed to production
- [] Ensure your migration is not going to run at peak time
- [] Run your migration: ``rake db:migrate:up VERSION=YYYYMMDDHHMMSS``
- [] Ensure ``db/seeds.rb`` has been updated to reflect the changes in the migration

The end

The end?

Not quite

Not quite

- Even checklists are fallible

Not quite

- Even checklists are fallible
- By the time the PR is opened the engineering work is finished

Not quite

- Even checklists are fallible
- By the time the PR is opened the engineering work is finished
- Still doesn't handle cases like a schema change taking 24 hours because of how large the table is

Not quite

- Even checklists are fallible
- By the time the PR is opened the engineering work is finished
- Still doesn't handle cases like a schema change taking 24 hours because of how large the table is
- Also doesn't handle people making mistakes!

We need to go deeper

Fixing issues before they happen

Fixing issues before they happen

- We should warn engineers about risks while they're writing their migrations

Fixing issues before they happen

- We should warn engineers about risks while they're writing their migrations
- That sounds like linting

Fixing issues before they happen

- We should warn engineers about risks while they're writing their migrations
- That sounds like linting
- Let's get rubocop on the case!

Let's take an example

```
class AddUrlToCommentAndMessage < ActiveRecord::Migration
  def up
    add_column :messages, :url, :text, :limit => 1024
    add_column :messages, :ua, :text, :limit => 1024
    add_column :comments, :url, :text, :limit => 1024
    add_column :comments, :ua, :text, :limit => 1024
    add_column :comments, :emailed, :boolean, :null => false, :default => false
  end

  def down
    remove_column :comments, :emailed
    remove_column :comments, :url
    remove_column :messages, :url
    remove_column :comments, :ua
    remove_column :messages, :ua
  end
end
```

What's wrong with it?

What's wrong with it?

- Each line executes a new SQL query

What's wrong with it?

- Each line executes a new SQL query
- Migration isn't atomic

What's wrong with it?

- Each line executes a new SQL query
- Migration isn't atomic
- Put this in your `.rubocop.yml`

```
Rails/BulkChangeTable:  
  Enabled: true  
  Database: mysql  
  Include:  
    - db/migrate/*
```

CustomCops/ OnlyAtomicMigrations

Intercom's own rubocop
rule for handling un-
bulked migrations



What else?

What else?

- BIGINT foreign keys

What else?

- BIGINT foreign keys
- Default for new tables is BIGINT primary keys

What else?

- BIGINT foreign keys
- Default for new tables is BIGINT primary keys
- Sometimes external references get the wrong type

CustomCops/ AlwaysMakeIdColumnsBigints

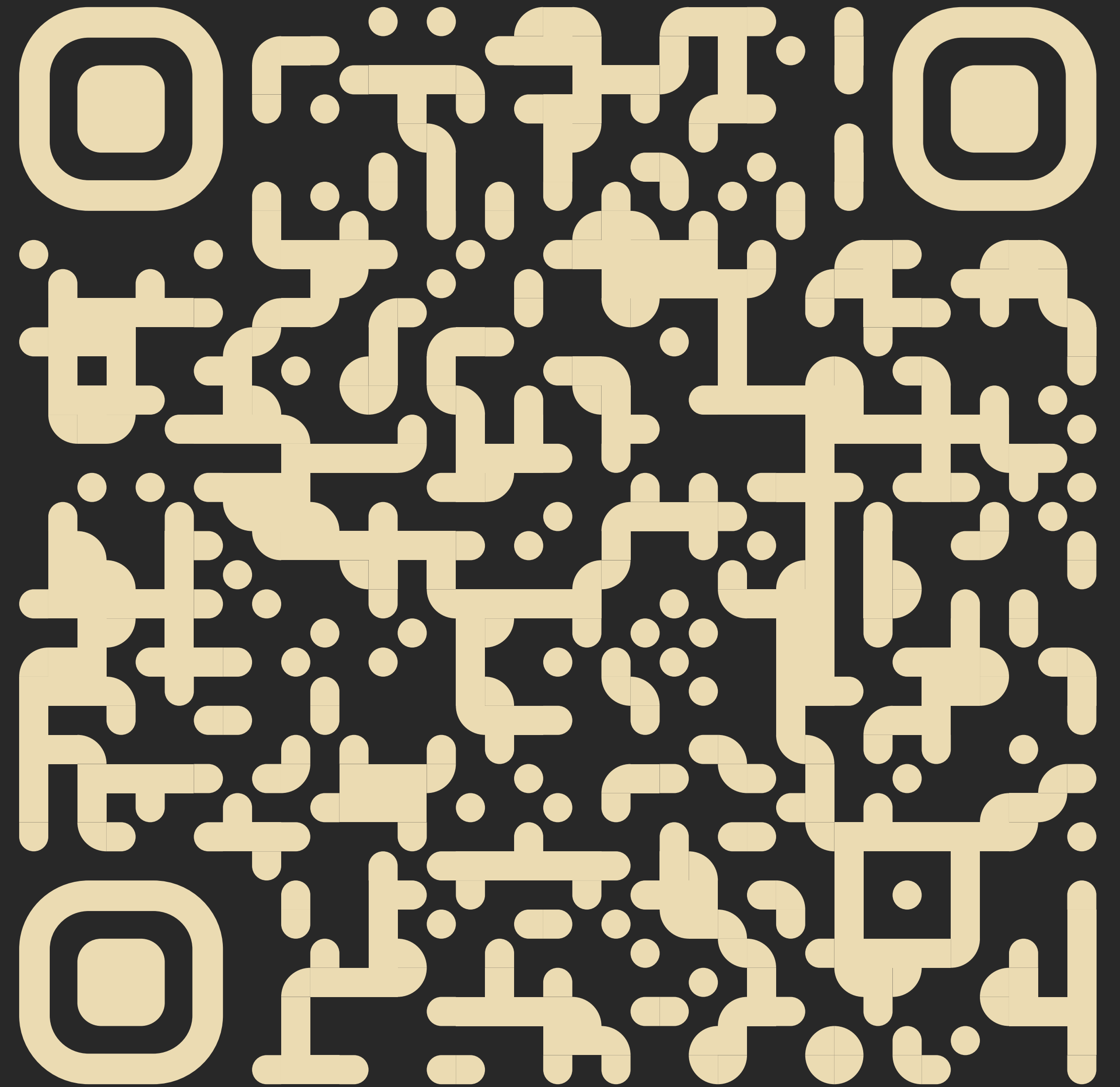
Block columns ending in
_id from being integer
typed



Last one

CustomCops/ NoExecuteInMigrations

Prevent using the
execute method in
migrations



Catching things during
development is great and
all...

Runtime checks

Unsafe operations

Unsafe operations

- Mistakes happen

Unsafe operations

- Mistakes happen
- What's safe in dev isn't necessarily safe in production

Unsafe operations

- Mistakes happen
- What's safe in dev isn't necessarily safe in production
- Risks are even greater now that LLMs are doing a lot of the driving

Block unsafe operations

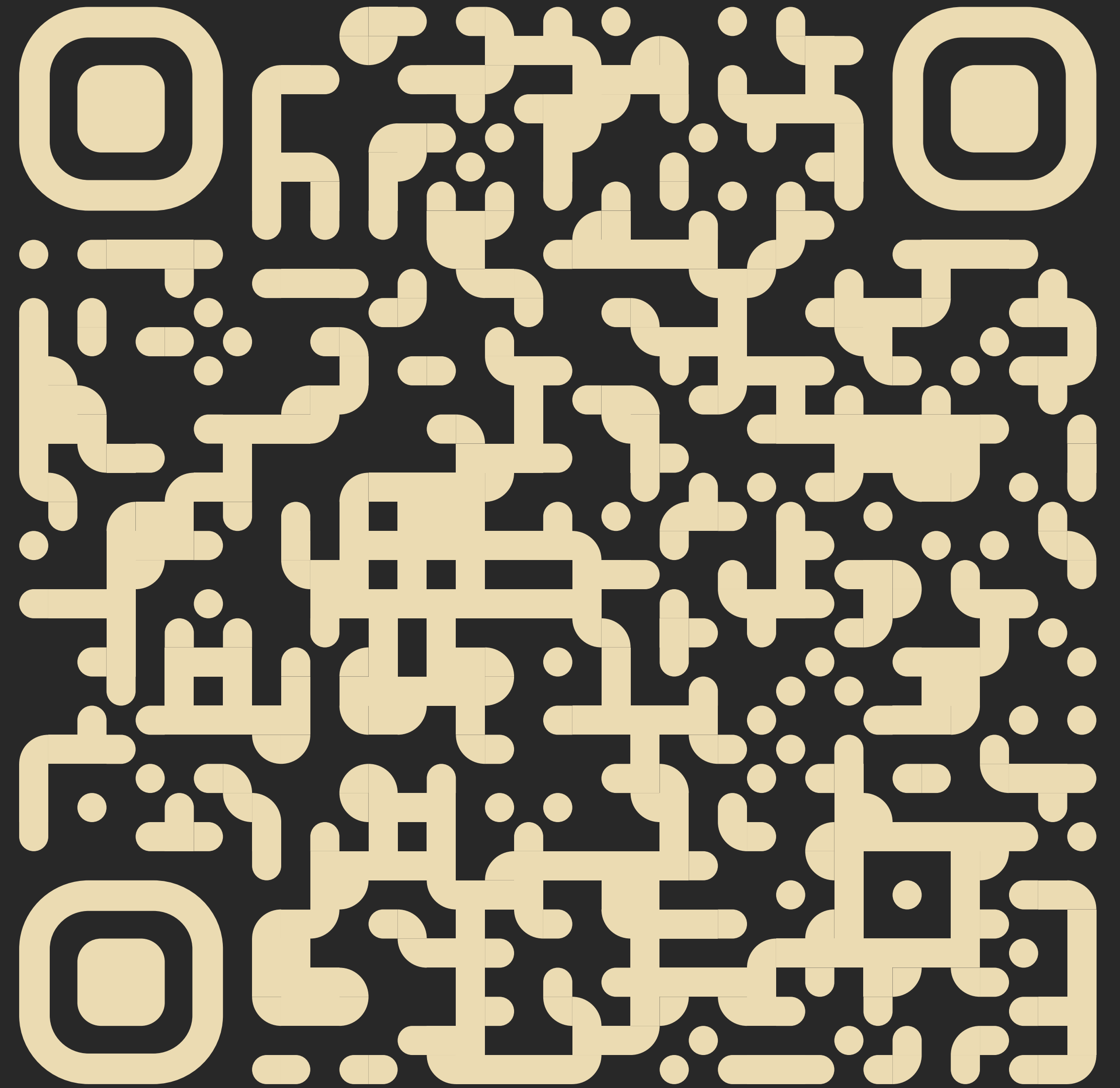
```
namespace :db do
  unless Rails.env.development? || Rails.env.test?
    tasks = Rake.application.instance_variable_get :@tasks
    tasks.delete "db:migrate"
    desc "db:migrate not available in this environment"
    task migrate: :environment do
      puts "db:migrate is not available in this environment, use db:migrate:up VERSION=YYYYMMDDHHMMSS"
    end

    tasks.delete "db:reset"
    desc "db:reset not available in this environment"
    task reset: :environment do
      puts "db:reset has been disabled"
    end

    tasks.delete "db:drop"
    desc "db:drop not available in this environment"
    task drop: :environment do
      puts "db:drop has been disabled"
    end
  end
end
end
```

Block unsafe operations

Override rake tasks in
your environment to stop
people taking unsafe
actions in production



Block unsafe migrations

Block unsafe migrations

- Some migrations just can't be safely executed directly

Block unsafe migrations

- Some migrations just can't be safely executed directly
- Tools like gh-ost exist and are excellent, but it's a lot to learn

Block unsafe migrations

- Some migrations just can't be safely executed directly
- Tools like gh-ost exist and are excellent, but it's a lot to learn
- Hard to know when you need to use a tool like that

Block unsafe migrations

- Some migrations just can't be safely executed directly
- Tools like gh-ost exist and are excellent, but it's a lot to learn
- Hard to know when you need to use a tool like that
- Breaks the “just be normal” principle we talked about earlier – can't just run migrations with rake.

Analysing migrations at runtime

Analysing migrations at runtime

- Monkey-patched ActiveRecord::Migration#exec_migration

Analysing migrations at runtime

- Monkey-patched ActiveRecord::Migration#exec_migration
- Used ActiveRecord::CommandRecorder to find out what the migration was going to execute

Analysing migrations at runtime

- Monkey-patched `ActiveRecord::Migration#exec_migration`
- Used `ActiveRecord::CommandRecorder` to find out what the migration was going to execute
- Implemented our heuristics in code

Analysing migrations at runtime

- Monkey-patched `ActiveRecord::Migration#exec_migration`
- Used `ActiveRecord::CommandRecorder` to find out what the migration was going to execute
- Implemented our heuristics in code
- Raise and block the migration if there's risk

Rules as code

```
if commands.any? { |operation, _| operation == :execute }
  raise(
    ActiveRecord::UnsafeDirectMigration,
    error_message(
      table: table_name,
      version: version,
      direction: direction,
      reason: "`execute` is never supported in migrations in Intercom"
    )
  )
end

if row_count >= ROW_COUNT_LIMIT || store_size >= STORE_SIZE_LIMIT
  raise(
    ActiveRecord::UnsafeDirectMigration,
    error_message(
      table: table_name,
      version: version,
      direction: direction,
      reason: unsafe_migration_reason(table_name, row_count, store_size)
    )
  )
end

super(conn, direction)
end

private def unsafe_migration_reason(table_name, row_count, store_size)
  return "`#{table_name}` is estimated to have #{number_to_human(row_count).downcase} rows which is greater than the cutoff
limit of #{number_to_human(ActiveRecord::DetectUnsafeMigrations::ROW_COUNT_LIMIT).downcase}" if row_count >= ROW_COUNT_LIMIT
  return "`#{table_name}` is estimated to occupy #{number_to_human_size(store_size)} which is greater than the cutoff limit of
#{number_to_human_size(ActiveRecord::DetectUnsafeMigrations::STORE_SIZE_LIMIT)}" if store_size >= STORE_SIZE_LIMIT
end
```

Getting the row count and size

```
private def get_table_row_count(conn, table)
  conn.exec_query("EXPLAIN SELECT COUNT(*) FROM #{table}").to_a.first["rows"] # innodb estimated cardinality of table
end

private def get_table_size(conn, table)
  # Estimated store size of the table and its indices (in bytes)
  query_result = conn.exec_query(<<~SQL.squish)
    SELECT (data_length + index_length) AS size
    FROM information_schema.tables
    WHERE table_schema = '#{conn.current_database}'
      AND table_name = '#{table}'
  SQL

  row = query_result.to_a.first
  row.present? ? row["size"] : 0
end
```

Making drop_table safe

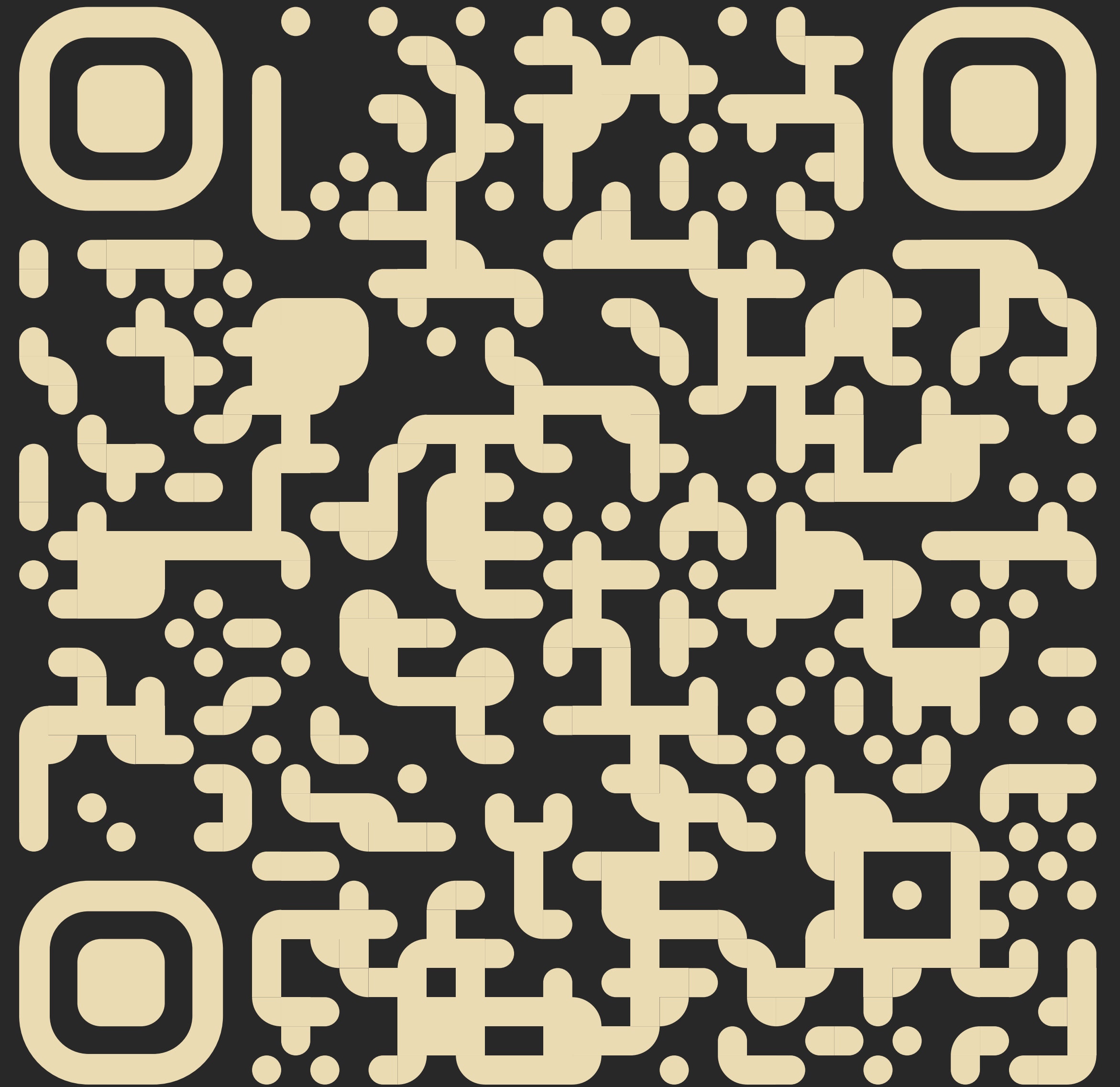
```
if commands.any? { |operation, _| operation == :drop_table } && row_count >= 2000
  self.instance_eval <<~RUBY
    def change
      execute "RENAME TABLE #{quote_table_name(table_name)} TO #{quote_table_name("_#{version}_#{table_name}_del")}"
    end
  RUBY
  return super(conn, direction)
end
```

Blocking change_column

```
if commands.any? { |operation, _| operation == :change_column } && row_count >= 2000
  raise(
    ActiveRecord::UnsafeDirectMigration,
    error_message(
      table: table_name,
      version: version,
      direction: direction,
      reason: "change_column is no longer supported for in-place migrations"
    )
  )
end
```

Detect unsafe migrations

Analyse migrations at runtime and block them based on heuristics



Future

Thanks !

Here's a link covering
all the code examples
and previous links from
this session.

