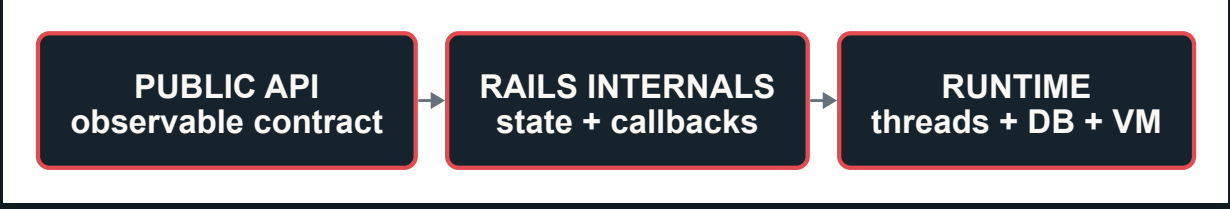
ΤΟΜΟΣ 2 · Η ΣΥΝΕΧΕΙΑ ΤΟΥ RAILS ΠΕΡΑ ΑΠΟ ΤΟ CRUD

Rails στα Έγκατα

Internals, runtime και failure mechanics πέρα από τα δημόσια APIs.



32

Από το API στον μηχανισμό

ΚΕΦΑΛΑΙΑ · RAILS 8.1.3.1 · RUBY 3.4.7

ΠΡΙΝ ΑΝΟΙΞΟΥΜΕ ΤΟ FRAMEWORK

Ο πρώτος τόμος βρήκε τα boundaries. Τώρα θα δούμε τι τα κρατά όρθια.

Το «Rails πέρα από το CRUD» τελείωσε με μια συγκεκριμένη ικανότητα: να βρίσκεις ποιο boundary κατέχει μια εγγύηση. Transaction, unique index, job contract, cache key, client protocol ή release overlap. Αυτός ο τόμος αρχίζει ακριβώς εκεί. Παίρνει κάθε boundary και ακολουθεί την εκτέλεση ως τον μηχανισμό.

Δεν είναι κατάλογος private methods. Τα internals αλλάζουν. Η αξία τους είναι ότι σε μαθαίνουν να σχηματίζεις και να ελέγχεις ένα causal model. Ποιο object κρατά state; Ποιος κατέχει το connection; Πότε γίνεται materialize μια transaction; Ποια row μετακινεί ένα job από scheduled σε ready; Τι ακριβώς παραμένει ζωντανό όταν ο Rack body κάνει stream;

Πώς διαβάζεται

Τα κεφάλαια δεν είναι ανεξάρτητα tips. Τα 1 έως 6 χτίζουν τον χάρτη εκτέλεσης. Τα 7 έως 16 ανοίγουν το Active Record. Τα 17 έως 26 ακολουθούν durable work και ασυνέπεια ανάμεσα σε συστήματα. Τα 27 έως 32 κατεβαίνουν στον Ruby runtime και επιστρέφουν σε ένα rolling release που πρέπει να αποδειχθεί ασφαλές.

Κάθε κεφάλαιο έχει `SOURCE TRACE` και `PROBE`. Το πρώτο δείχνει την ακριβή διαδρομή μέσα στην έκδοση αναφοράς. Το δεύτερο μετατρέπει το mental model σε παρατηρήσιμο πείραμα. Δεν χρειάζεται να θυμάσαι private names. Χρειάζεται να ξέρεις πώς θα τα ξαναβρείς.

Έκδοση αναφοράς

Η έκδοση 1.0 ελέγχθηκε τον Σεπτέμβριο του 2026 με Rails 8.1.3.1, Ruby 3.4.7, Solid Queue 1.5.0, Rack 3.2.7 και Puma 8.0.2. Τα source traces κοιτούν αυτές τις εκδόσεις. Όπου ένα mechanism είναι private, επισημαίνεται ρητά. Όπου η συμπεριφορά ανήκει στον adapter, στη database ή στη VM, δεν αποδίδεται στο Rails.

Το βιβλίο δεν προτείνει να καλείς internals από application code. Προτείνει να τα διαβάζεις όταν η public συμπεριφορά δεν αρκεί για να εξηγήσει το production.

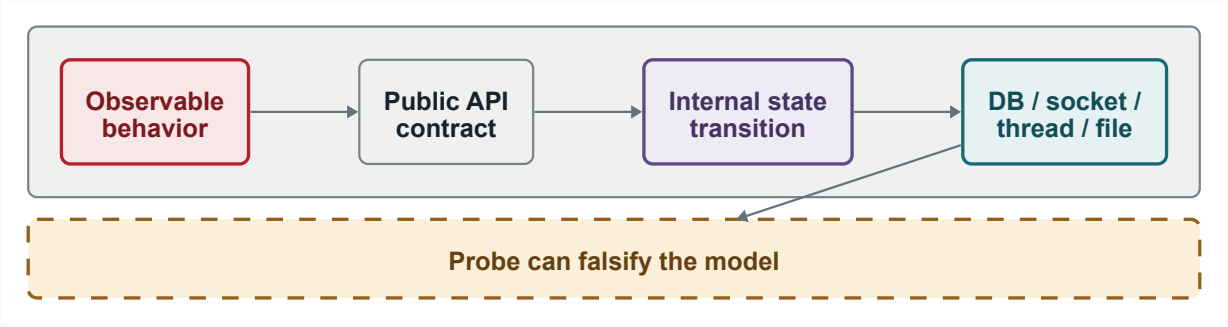
Ο ΧΑΡΤΗΣ ΤΟΥ ΤΟΜΟΥ 2

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1

Ο source ως εκτελέσιμος χάρτης

Το public API λέει τι μπορείς να ζητήσεις. Ο source δείχνει ποιο state αλλάζει, ποιος καθαρίζει μετά και πού ακριβώς μπορεί να κοπεί η εκτέλεση.



Σχήμα 1.1 · Η διαδρομή από observable behavior σε mechanism και probe.

Ο πρώτος τόμος τελείωσε με μια ερώτηση: ποιο boundary κρατά την εγγύηση; Τώρα η ερώτηση γίνεται πιο αυστηρή. Ποιο object κρατά το state αυτού του boundary, ποια methods το μετακινούν και ποιο γεγονός μπορεί να παρατηρήσει ένα test;

Αυτό δεν σημαίνει ότι η εφαρμογή πρέπει να καλεί private Rails APIs. Το αντίθετο. Το public API παραμένει το integration boundary. Διαβάζουμε τον source για να εξηγήσουμε behavior, να βρούμε το σωστό probe και να καταλάβουμε τι μπορεί να αλλάξει σε upgrade. Ένα private class name είναι συντεταγμένη στον τρέχοντα χάρτη, όχι contract που δικαιούται να κρατήσει το application code.

Τέσσερις στρώσεις μιας ερώτησης

Πάρε το `Reservation.where(status: :held).load_async`. Η observable ερώτηση είναι αν το query τρέχει παράλληλα με άλλη δουλειά. Το public contract βρίσκεται στο `Relation#load_async` και στη ρύθμιση του async query executor. Η υλοποίηση περνά από `Relation#exec_main_query`, `select_all(async: true)`, ένα `FutureResult` και τον executor του connection pool. Το probe δεν είναι να ελέγξει το όνομα `FutureResult`. Είναι να μετρήσει πότε αποκτάται connection, πόσα queries βρίσκονται ταυτόχρονα σε πτήση και αν το latency όντως επικαλύπτεται.

Στρώση	Ερώτηση	Σταθερότητα
Behavior	Τι βλέπει ο caller;	υψηλή
Public API	Τι υπόσχεται το Rails;	versioned
Mechanism	Ποια objects το υλοποιούν τώρα;	χαμηλότερη

Στρώση	Ερώτηση	Σταθερότητα
Probe	Ποια μέτρηση διαψεύδει το μοντέλο;	δική μας

Αν ξεκινήσεις από το private constant, κινδυνεύεις να περιγράψεις trivia. Αν ξεκινήσεις από observable behavior, ο source αποκτά κατεύθυνση. Ψάχνεις για την πρώτη public method, ακολουθείς μόνο calls που αλλάζουν relevant state και σταματάς όταν φτάσεις σε resource boundary: socket, database call, queue row, mutex, thread, fiber ή filesystem operation.

Η τεχνική του μικρού trace

Ένα καλό trace χωρά συνήθως σε πέντε έως οκτώ βήματα. Γράφει receiver, method, state transition και cleanup. Για ένα request, για παράδειγμα:

OUTPUT

```
ActionDispatch::Executor#call
  -> executor.run!
  -> app.call(env)
  -> Rack response body
  -> body.each
  -> body.close
  -> execution.complete!
```

Το τελευταίο βήμα είναι συχνά πιο σημαντικό από το πρώτο. Αν χαθεί το cleanup, μένουν checked out connections, execution-local state ή προσωρινές caches. Γι' αυτό κάθε trace αυτού του βιβλίου έχει δύο κατευθύνσεις: πώς μπαίνει η δουλειά και πώς βγαίνει, ακόμη όταν σηκωθεί exception ή κλείσει πρόωρα ένα stream.

Μην εμπιστεύεσαι μόνο το filename. Στο Rails ένα behavior συχνά σχηματίζεται από modules, callbacks και adapter overrides. Χρησιμοποίησε `source_location` για τη method που όντως έχει το runtime object και μετά δες τους ancestors.

RUBY

```
relation = Reservation.where(status: :held)

owner = relation.method(:load_async).owner
file, line = relation.method(:load_async).source_location

puts owner
puts "#{file}:#{line}"
puts relation.class.ancestors.first(8)
```

Η `method(:x).owner` απαντά ποιο module κέρδισε τη lookup. Το `source_location` απαντά πού ορίστηκε η Ruby method. Δεν βλέπει C methods και δεν αποδεικνύει ποιο branch θα τρέξει. Είναι η αρχή του trace, όχι το τέλος του.

Contract test, characterization test, source note

Κράτα τρία διαφορετικά artifacts. Ένα contract test ελέγχει τη συμπεριφορά που χρειάζεται η εφαρμογή και πρέπει να επιβιώσει από refactor. Ένα characterization test καταγράφει δύσκολη

τρέχουσα συμπεριφορά πριν από upgrade. Ένα source note κρατά paths και line ranges για να ξαναβρείς γρήγορα τον μηχανισμό. Μην κάνεις assert private class names μέσα στο κανονικό test suite, εκτός αν έχεις επιλέξει συνειδητά να συντηρείς framework extension.

SOURCE TRACE

Η έκδοση αναφοράς βρίσκεται στο tag `v8.1.3.1`. Ξεκίνα από την public method, χρησιμοποίησε το runtime `source_location` και επιβεβαίωσε το αποτέλεσμα στο αντίστοιχο tagged αρχείο. Για το παράδειγμα, η είσοδος είναι το `activerecord/lib/active_record/relation.rb` και η async εκτέλεση συνεχίζει στα `future_result.rb` και `connection_adapters/abstract/database_statements.rb`.

ΠΑΓΙΔΑ

Το ότι ένα internal object υπάρχει σε τρεις Rails εκδόσεις δεν το μετατρέπει σε public contract. Αν πρέπει να το αγγίξεις, απομόνωσε τον adapter, έλεγξε `respond_to?` ή method parameters και βάλε ένα upgrade characterization test.

ΜΗΧΑΝΙΣΜΟΣ

Το source reading είναι causal debugging. Ξεκινά από behavior, ακολουθεί state και resource ownership και τελειώνει σε probe που μπορεί να κάνει το μοντέλο λάθος.

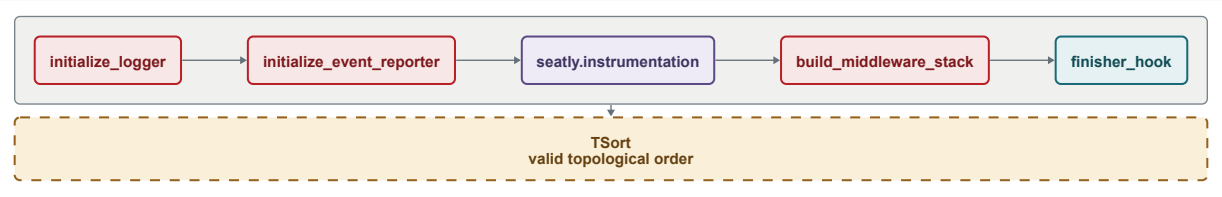
PROBE

Άνοιξε `bin/rails console`, διάλεξε μία Rails method που χρησιμοποιείς καθημερινά και τύπωσε `owner`, `source_location` και τους πρώτους ancestors του receiver. Γράψε trace μέχρι το πρώτο I/O boundary. Σημείωσε επίσης ποια method κάνει το cleanup. Αν δεν τη βρεις, το trace δεν έχει τελειώσει.

ΚΕΦΑΛΑΙΟ 2

Το initializer DAG

Το boot δεν είναι μια μακριά λίστα callbacks. Είναι γράφος εξαρτήσεων που το Rails ταξινομεί, δένει πάνω στην εφαρμογή και εκτελεί μία φορά.



Σχήμα 2.1 · Τα before και after constraints μετατρέπονται σε τοπολογική σειρά.

Όταν γράφεις `initializer "seatly.metrics", after: :initialize_logger`, δεν ζητάς απλώς μια θέση σε array. Δηλώνεις ακμή σε directed graph. Το Rails συγκεντρώνει initializers από railties, engines και application ancestors, μετατρέπει `before` και `after` σε dependencies και χρησιμοποιεί το `TSort` της Ruby για topological sort. Αυτό εξηγεί γιατί η σειρά παραμένει σταθερή όταν προστεθεί ένα άσχετο engine, αλλά επίσης γιατί ένας κύκλος δεν μπορεί να «λυθεί» με load order.

Από δήλωση σε κόμβο

Το `Rails::Initializable::Initializer` κρατά `name`, `before`, `after`, `group`, context και block. Η δήλωση σε class level δημιουργείται αρχικά χωρίς bound context. Κατά το boot, το `initializers_for(self)` αντιγράφει τη chain και δένει κάθε initializer στο συγκεκριμένο application instance. Έτσι το block εκτελείται με `instance_exec` και βλέπει το σωστό config.

Η `Collection#<<` χτίζει δύο maps. Το `@order` κρατά ποια ονόματα πρέπει να προηγηθούν ενός άλλου. Το `@resolve` αντιστοιχίζει name σε πραγματικούς initializer objects. Το `tsort_each_child` χρησιμοποιεί αυτά τα maps για να δώσει στον `TSort` τις ακμές. Η default συμπεριφορά, όταν δεν δίνεται constraint, είναι να τοποθετηθεί ο νέος initializer μετά από τον τελευταίο γνωστό. Είναι βολικό, αλλά δεν είναι καλό contract για extension που εξαρτάται από συγκεκριμένο framework state.

RUBY

```ruby
module Seatly
  class Engine < ::Rails::Engine
    initializer "seatly.instrumentation",
      after: :initialize_event_reporter do
      ActiveSupport::Notifications.subscribe("seatly.reserve") do |event|
        Rails.logger.info(duration_ms: event.duration)
      end
    end
  end
end
```

Το constraint πρέπει να εκφράζει την πραγματική προϋπόθεση. Αν ο subscriber χρειάζεται logger και event reporter, διάλεξε το αργότερο relevant boundary. Μην χρησιμοποιείς ένα initializer name που απλώς έτυχε να βρίσκεται κοντά στη σημερινή σειρά. Τα names είναι πιο σταθερά από line positions, όχι απαραίτητα public API για πάντα.

Bootstrap, railties, finisher

Η application initializer chain περιλαμβάνει τρεις νοητικές φάσεις. Το `Rails::Application::Bootstrap` στήνει πρώιμα στοιχεία όπως logger, error reporter, event reporter, cache και τον `once` autoloader. Ανάμεσα βρίσκεται η αλυσίδα των railties και engines. Το `Rails::Application::Finisher` στήνει τον main autoloader, middleware, prepare callbacks, eager loading, executor hooks, routes reloaders και το τελικό `after_initialize` hook.

Οι φάσεις δεν είναι ξεχωριστά scheduler passes. Συνεισφέρουν initializers στην ίδια chain και τα constraints ορίζουν την τελική σειρά. Το `group` επιτρέπει διαφορετικό subset, όπως `:all`, αλλά το συνηθισμένο boot τρέχει την `:default` ομάδα. Η `run_initializers` φυλά ένα `@ran`, άρα το ίδιο application instance δεν επαναλαμβάνει τη διαδικασία.

Αυτή η one-shot ιδιότητα αλλάζει το design. Κώδικας που πρέπει να ανανεώνεται στο development δεν ανήκει απλώς σε initializer. Χρειάζεται `to_prepare` ή autoload hook με idempotent behavior. Διαφορετικά το initializer κρατά class object που θα αποσυνδεθεί μετά από reload.

Να βλέπεις τον γράφο

Το Rails δεν δίνει έτοιμο production graph visualization, αλλά η chain είναι inspectable. Μπορείς να τυπώσεις names και constraints χωρίς να εκτελέσεις δεύτερο boot.

```ruby
rows = Rails.application.initializers.map do |item|
  [item.name, item.before, item.after, item.context_class.name]
end

rows.each { |row| puts row.compact.join(" | ") }
```

Αν δύο gems συγκρούονται, ψάξε τον μικρό υπογράφο γύρω από τα relevant names. Μην προσπαθήσεις να εξηγήσεις εκατοντάδες κόμβους. Βρες το state που λείπει, ποιος initializer το παράγει και ποιος το καταναλώνει. Ένας κύκλος σημαίνει ότι οι δηλωμένες προϋποθέσεις είναι ασύμβατες. Η λύση είναι να αλλάξεις το boundary ή να χωρίσεις έναν initializer, όχι να ελπίζεις σε άλλη σειρά require.

Names, groups και κρυφές ακμές

Το `before` και το `after` μπορούν να αναφέρονται σε name που δεν υπάρχει στη συγκεκριμένη εφαρμογή. Αυτό δεν είναι απαραίτητα σφάλμα. Επιτρέπει σε railtie να δηλώνει συμβατό ordering χωρίς hard dependency στο άλλο component. Σημαίνει όμως ότι το effective graph αλλάζει όταν προστεθεί ή αφαιρεθεί gem. Κράτα snapshot των initializer names σε framework upgrades και σύγκρινε τις ακμές γύρω από τα δικά σου extension points.

Τα initializer groups είναι δεύτερος άξονας. Ένα initializer μπορεί να ανήκει στο default run αλλά να λείπει από άλλο group που χρησιμοποιεί task ή custom runner. Αν command δουλεύει στον web boot

και αποτυγχάνει σε maintenance task, μην κοιτάς μόνο τη σειρά. Έλεγξε αν εκτελέστηκε ο κόμβος. Το σωστό diagnostic artifact περιέχει name, context class, group και resolved predecessors. Έτσι μια τυχαία boot διαφορά γίνεται μικρό, ελέγξιμο graph diff.

SOURCE TRACE

Ο μηχανισμός βρίσκεται στο `railties/lib/rails/initializable.rb`. Οι αρχικές και τελικές φάσεις βρίσκονται στα `application/bootstrap.rb` και `application/finisher.rb`. Ακολούθησε `initializer -> Collection#<< -> tsort_each -> Initializer#run`.

ΠΑΓΙΔΑ

Μη βάζεις application records ή reloadable classes σε initializer state. Στο development μπορεί να κρατήσεις stale class object. Στο production μπορεί να ανοίξεις connection πριν ολοκληρωθεί σωστά το pool configuration.

ΜΗΧΑΝΙΣΜΟΣ

Ένας initializer είναι κόμβος με prerequisites και one-shot lifecycle. Η σωστή ερώτηση δεν είναι «πού να τον βάλω;» αλλά «ποιο state απαιτεί και ποιος το έχει ήδη παράγει;».

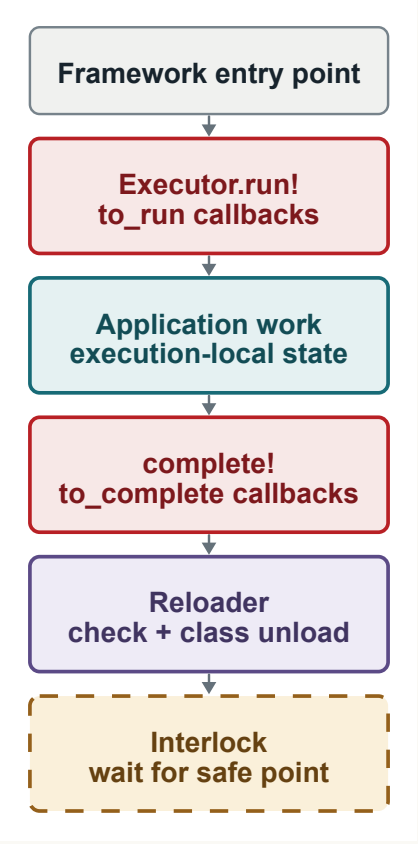
PROBE

Τύπωσε τους initializers που έχουν `autoload`, `middleware`, `active_record` ή το όνομα ενός engine σου. Κατέγραψε `before` και `after`. Έλεγξε αν ο δικός σου initializer δηλώνει πραγματική εξάρτηση ή βασίζεται στην τυχαία προηγούμενη θέση του. Τρέξε το ίδιο probe πριν και μετά από framework upgrade.

ΚΕΦΑΛΑΙΟ 3

Executor, Reloader και Interlock

To execution context δεν είναι thread local από συνήθεια. Είναι lifecycle με είσοδο, cleanup και συντονισμό με code reload.



Σχήμα 3.1 · Ο Executor τυλίγει application work και ο Reloader συντονίζει το unload.

Το Rails καλεί application code από πολλά entry points: HTTP requests, jobs, Cable message handlers και command runners. Σε κάθε entry πρέπει να ενεργοποιήσει προσωρινό state και να το καθαρίσει. Ο `ActiveSupport::ExecutionWrapper` δίνει αυτό το γενικό πρωτόκολλο. Η subclass που εκθέτει η εφαρμογή ως executor έχει `to_run` και `to_complete` callbacks. Οι callbacks ενεργοποιούν μεταξύ άλλων query cache lifetimes, connection cleanup και ασφαλή θέση απέναντι στο reload.

Η συμμετρία run και complete

Το `run!` δημιουργεί instance, το αποθηκεύει στο `IsolatedExecutionState` και τρέχει τη `:run` callback chain. Το `complete!` τρέχει τη δεύτερη chain και διαγράφει το active key σε `ensure`. Η `wrap` βάζει

αυτά τα δύο γύρω από block, αναφέρει unhandled exception μέσω error reporter και ξανασηκώνει το exception.

Η wrapper είναι re-entrant. Αν είναι ήδη active στο τρέχον execution context, η `wrap` κάνει yield χωρίς δεύτερη callback pair. Αυτό αποτρέπει διπλό cleanup όταν framework component καλέσει άλλο wrapped component. Δεν σημαίνει ότι ένα καινούριο thread κληρονομεί το context. Το νέο thread χρειάζεται δικό του wrap.

```ruby
class SeatlyPoller
  def start
    Thread.new do
      loop do
        Rails.application.executor.wrap do
          Reservation.expired.limit(100).find_each(&:release!)
        end
        sleep 1
      end
    end
  end
end
```

Ο κώδικας αυτός δείχνει το lifecycle, όχι ότι πρέπει να αντικαταστήσεις ένα job backend με χειροποίητο poller. Το κρίσιμο σημείο είναι ότι κάθε iteration, όχι ολόκληρο το άπειρο loop, αποκτά και ολοκληρώνει execution context. Αλλιώς ένα connection ή cache μπορεί να ζει όσο η διεργασία.

Γιατί υπάρχει Reloader

Ο Reloader είναι specialized `ExecutionWrapper` για top-level loops που ξαναμπαίνουν σε application code. Ελέγχει αν άλλαξαν watched files, περιμένει ασφαλές σημείο, εκτελεί class unload και τρέχει `to_prepare`. Στο production, όπου reloading είναι disabled, λειτουργεί ουσιαστικά ως πέρασμα προς τον Executor. Στο development ο συγχρονισμός είναι ουσιαστικός: η σταθερά `User` πριν και μετά το reload μπορεί να δείχνει διαφορετικό class object.

Το Interlock κρατά modes για threads που τρέχουν application code, κάνουν load ή κάνουν unload. Ένα reload πρέπει να περιμένει να φύγουν οι active executions. Γι' αυτό child thread δεν πρέπει να καλέσει Reloader ενώ ο parent παραμένει μέσα σε Executor και περιμένει το child. Ο child ζητά unload, το unload περιμένει τον parent και ο parent περιμένει το child. Αυτός είναι πραγματικός deadlock.

Για προσωρινή παράλληλη εργασία μέσα σε wrapped code, η επίσημη escape hatch είναι `permit_concurrent_loads` γύρω από το blocking join. Η χρήση του δηλώνει ότι το block δεν θα autoload application constants. Αν αυτή η δήλωση είναι λάθος, αφαιρείς την προστασία ακριβώς όταν τη χρειάζεσαι.

```ruby
thread = Thread.new do
  Rails.application.executor.wrap { ExternalProbe.call }
end

ActiveSupport::Dependencies.interlock.permit_concurrent_loads do
  thread.join
end
```

IsolatedExecutionState δεν σημαίνει process global

Το execution state μπορεί να απομονώνεται ανά thread ή ανά fiber, ανάλογα με τη ρύθμιση. Εκεί αποθηκεύονται context stacks όπως current database role και ενεργός execution wrapper. Η σωστή νοητική εικόνα δεν είναι «ένα hash σαν global». Είναι state που ανήκει στην τρέχουσα μονάδα εκτέλεσης και πρέπει να δημιουργείται και να καθαρίζεται στα framework entry points.

Αν μια βιβλιοθήκη καλεί callbacks της εφαρμογής από δικό της thread pool, εκείνη γίνεται entry point και αναλαμβάνει το wrap. Αν απλώς εκτελεί pure Ruby ή I/O χωρίς application constants και Active Record, ίσως δεν χρειάζεται. Η απόφαση βασίζεται στο ποιος καλεί application code, όχι στο όνομα του thread.

SOURCE TRACE

Ακολούθησε `ExecutionWrapper.run!`, `wrap` και `complete!`, μετά τον `ActiveSupport::Reloader` και το `Dependencies::Interlock`. Το επίσημο contract και το child-thread deadlock περιγράφονται στο `Threading and Code Execution`.

ΠΑΓΙΔΑ

Μην αντιγράφεις execution-local state χειροκίνητα σε thread ή fiber. Πέρασε τα domain values ρητά και άφησε το entry point να δημιουργήσει σωστό Rails context. Η τυφλή αντιγραφή μπορεί να μεταφέρει λάθος tenant, role ή connection ownership.

ΜΗΧΑΝΙΣΜΟΣ

Ο Executor ορίζει πότε application state είναι ενεργό. Ο Reloader ορίζει πότε μπορεί να αντικατασταθεί application code. Το Interlock εμποδίζει αυτές τις δύο αλήθειες να συγκρουστούν.

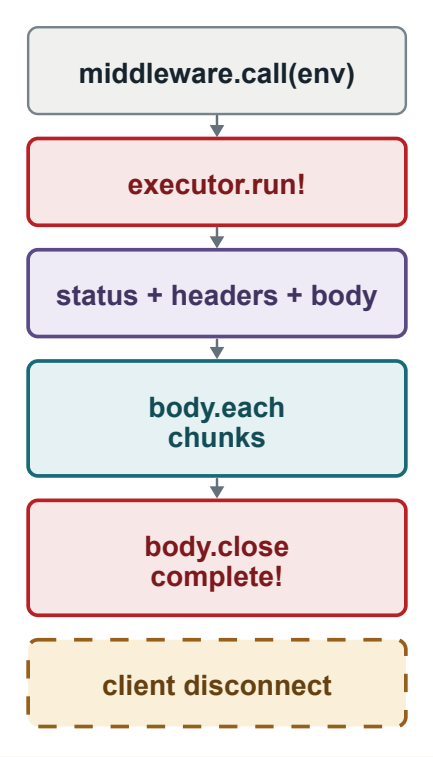
PROBE

Γράψε ένα μικρό development probe που τρέχει application query σε νέο thread με και χωρίς `executor.wrap`. Μέτρησε το `ActiveRecord::Base.connection_pool.stat` πριν και μετά. Μετά άλλαξε ένα model file κατά την εκτέλεση. Μη βασιστείς μόνο στο αν «δούλεψε». Κατέγραψε cleanup και reload behavior.

ΚΕΦΑΛΑΙΟ 4

Το lifecycle του Rack body

Η επιστροφή του τριπλέτου Rack δεν σημαίνει ότι τελείωσε το request. Για streaming response, η πραγματική έξοδος βρίσκεται στο `body.close`.



Σχήμα 4.1 · Το middleware ξετυλίγεται, αλλά το cleanup μπορεί να περιμένει το body close.

Το Rack endpoint επιστρέφει status, headers και body. Το body είναι iterable object που παράγει chunks και μπορεί να έχει `close`. Σε απλή απόκριση τα chunks βρίσκονται ήδη σε array. Σε streaming ή file response η κατανάλωση συνεχίζεται αφού το `call` έχει επιστρέψει. Αυτό αναγκάζει middleware που κρατά lifecycle state να μεταφέρει το cleanup πάνω στο body.

Το `ActionDispatch::Executor#call` είναι καθαρό παράδειγμα. Καλεί `executor.run!`, εκτελεί το εσωτερικό app και δίνει το result στο `Rack::BodyProxy`, με callback που καλεί `state.complete!`. Αν το inner app σηκώσει exception πριν δώσει response, το middleware ολοκληρώνει το state αμέσως. Αν επιστρέψει body, η ολοκλήρωση αναβάλλεται μέχρι να κλείσει το proxy. Αυτό διατηρεί το Rails execution context ενεργό όσο η εφαρμογή παράγει chunks.

Headers committed, body sending, body sent

Το `ActionDispatch::Response` έχει ξεχωριστές καταστάσεις. Το `commit!` οριστικοποιεί headers και ξυπνά threads που περιμένουν το commit. Το `each` καλεί `sending!`, διατρέχει το stream και μετά `sent!`. Αυτά δεν είναι συνώνυμα. Μετά το commit δεν μπορείς να αλλάξεις αξιόπιστα status ή headers, αλλά το body μπορεί ακόμη να παράγεται για πολλή ώρα.

Σε `ActionController::Live`, άλλο thread μπορεί να γράφει στο response stream. Το πρώτο `write` κάνει commit. Από εκεί και πέρα ένα exception δεν μπορεί να μετατραπεί σε καθαρό JSON 500, επειδή bytes και headers έχουν ήδη φύγει. Το failure contract γίνεται διακοπή stream, trailer όπου υποστηρίζεται ή application-level frame που ο client καταλαβαίνει.

RUBY

```ruby
class ExportsController < ApplicationController
  include ActionController::Live

  def show
    response.headers["Content-Type"] = "text/csv"
    response.stream.write("reservation_id,status\n")
    Reservation.find_each do |reservation|
      response.stream.write("#{reservation.id},#{reservation.status}\n")
    end
  ensure
    response.stream.close
  end
end
```

Το `ensure` είναι μέρος του correctness, όχι style. Ο server ή middleware θα προσπαθήσει να κλείσει bodies, αλλά ο producer κατέχει δικούς του πόρους και πρέπει να έχει explicit cleanup. Αν ο client αποσυνδεθεί, το write μπορεί να σηκώσει I/O error. Το `ensure` πρέπει ακόμη να τρέξει.

Middleware και body proxies

Ένα συνηθισμένο custom middleware μετρά μόνο τη διάρκεια του `app.call`. Για streaming endpoint αυτό μετρά time to headers, όχι συνολική διάρκεια ή bytes. Αν χρειάζεσαι end-to-end timing, τύλιξε το body και ολοκλήρωσε τη μέτρηση στο `close`. Αν χρειάζεσαι time to first byte και stream duration, κράτα δύο metrics.

RUBY

```ruby
class BodyTiming
  def initialize(app)
    @app = app
  end

  def call(env)
    started = Process.clock_gettime(Process::CLOCK_MONOTONIC)
    status, headers, body = @app.call(env)
    callback = proc { record(started, env) }
    [status, headers, Rack::BodyProxy.new(body, &callback)]
  rescue Exception
    record(started, env)
    raise
  end

  private
```

```
  def record(started, env)
    elapsed = Process.clock_gettime(Process::CLOCK_MONOTONIC) - started
    Rails.event.notify("seatly.response.closed", duration: elapsed,
      path: env["PATH_INFO"])
  end
end
```

Το callback πρέπει να είναι idempotent ή να βασίζεται στην εγγύηση του proxy ότι εκτελείται μία φορά. Πρόσεξε και την ιδιοκτησία του αρχικού body: το proxy πρέπει να προωθεί `close`, όχι να το καταπίνει.

Το κρυφό capacity κόστος

Ένα αργό stream μπορεί να κρατά execution context, controller objects και ίσως database connection αν η παραγωγή κάνει lazy queries. Το ότι επιστράφηκαν headers δεν ελευθερώνει worker capacity. Αποσύνδεσε database work από αργό network writing: φόρτωσε σε bounded batches, μην κρατάς transaction και μην υποθέτεις ότι `find_each` σημαίνει μηδενική χρήση connection.

Disconnect, hijack και cancellation

Ένα client disconnect δεν φτάνει πάντα ως καθαρό exception στο σημείο όπου παράγεις chunks. Μπορεί να εμφανιστεί στο επόμενο write, στο server adapter ή μόνο όταν κλείσει το body. Γι’ αυτό η cancellation πρέπει να είναι cooperative: κάθε bounded βήμα ελέγχει αν μπορεί να συνεχίσει και το `ensure` απελευθερώνει ό,τι απέκτησε. Δεν βασίζεται σε ένα συγκεκριμένο exception class.

Το Rack hijack παραδίδει χαμηλότερου επιπέδου I/O στην εφαρμογή και αλλάζει ακόμη περισσότερο την ιδιοκτησία. Από εκεί και πέρα το συνηθισμένο response body contract δεν περιγράφει όλο το lifetime. Middleware που μετρά duration ή κρατά resource μέχρι `BodyProxy#close` χρειάζεται ξεχωριστή απόφαση για hijacked responses. Πριν υποστηρίξεις τέτοιο path, γράψε πίνακα με owner για socket, execution wrapper, database lease και cleanup callback. Αν ένα κελί δεν έχει μοναδικό owner, έχεις leak που απλώς δεν έχει συμβεί ακόμη.

Response protocol πριν και μετά το commit

Early Hints, file sending και server-specific acceleration δείχνουν γιατί το Rack tuple είναι μόνο το κοινό επίπεδο. Ένα middleware μπορεί να προσθέσει hint πριν από το τελικό response, ενώ άλλο μπορεί να αντικαταστήσει Ruby iteration με web-server file transfer. Η εφαρμογή πρέπει να μετρά observable lifecycle, όχι να συμπεραίνει ότι όλα τα bytes πέρασαν από το ίδιο `each`.

Χώρισε το response σε τρεις ζώνες. Πριν από header commit μπορείς ακόμη να αλλάξεις status και να επιστρέψεις κανονικό error document. Μετά το commit αλλά πριν από το τελευταίο byte μπορείς μόνο να ολοκληρώσεις ή να κόψεις το υπάρχον protocol. Μετά το close μπορείς να κάνεις cleanup και observation, όχι να διορθώσεις output. Αυτός ο διαχωρισμός βοηθά και σε non-streaming endpoints, επειδή middleware compression ή proxy buffering μπορεί να μεταθέσει το σημείο που ο client βλέπει bytes.

Για κάθε streaming route κατέγραψε first-byte latency, stream duration, bytes, termination reason και body-close latency ως διαφορετικά πεδία. Ένα ενιαίο request duration κρύβει αν το πρόβλημα ήταν controller work, αργός consumer ή cleanup που μπλόκαρε μετά την αποστολή.

SOURCE TRACE

Η είσοδος βρίσκεται στο `ActionDispatch::Executor#call`. Το state machine της απόκρισης βρίσκεται στο `ActionDispatch::Response`. Το close callback υλοποιείται από το `Rack::BodyProxy`.

ΠΑΓΙΔΑ

Μετά το πρώτο committed byte δεν έχεις πια το συνηθισμένο controller error contract. Σχεδίασε errors, cancellation και partial output ως streaming protocol. Ένα `rescue_from` δεν μπορεί να πάρει πίσω bytes από το socket.

ΜΗΧΑΝΙΣΜΟΣ

Το request lifetime για Rack τελειώνει όταν κλείσει το body. Όποιος τυλίγει resource γύρω από `call` πρέπει να αποφασίσει αν το resource ανήκει ως τα headers ή ως το τελευταίο chunk.

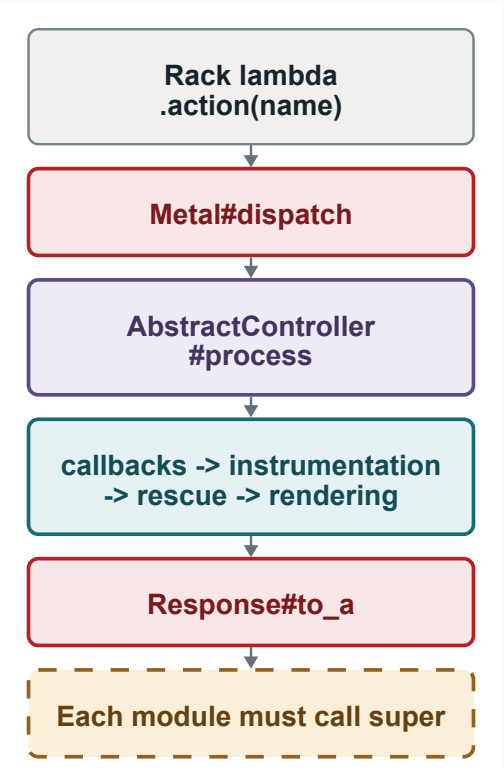
PROBE

Φτιάξε endpoint που στέλνει τρία chunks με μικρή καθυστέρηση. Βάλε timestamps πριν και μετά το `app.call`, στο πρώτο `each` και στο `close`. Κάνε request που ολοκληρώνεται και request που ο client κόβει νωρίς. Επιβεβαίωσε ποια callbacks τρέχουν και αν το connection pool επιστρέφει στο αρχικό busy count.

ΚΕΦΑΛΑΙΟ 5

Η αλυσίδα του controller

Το `ApplicationController` μοιάζει με μία class. Στην πράξη είναι σύνθεση modules που μετατρέπουν ένα Rack endpoint σε callback, rendering και error pipeline.



Σχήμα 5.1 · Το ActionController::Metal αποκτά συμπεριφορά από ordered modules.

Στη βάση υπάρχει το `ActionController::Metal`. Η class method `action(name)` επιστρέφει Rack lambda. Η lambda φτιάχνει `ActionDispatch::Request`, δημιουργεί response, κάνει `new.dispatch` και τελικά επιστρέφει `response.to_a`. Η `ActionController::Base` προσθέτει modules για strong parameters, rendering, redirects, callbacks, rescue, instrumentation, cookies, forgery protection και άλλα. Η σειρά inclusion είναι συμπεριφορά, επειδή αλλάζει τη method lookup chain και τα wrappers γύρω από `process_action`.

Η στενή μέση: process_action

Το `AbstractController::Base#process` βρίσκει το action method και περνά στο `process_action`. Διαφορετικά modules κάνουν override με `super`. Το callbacks module τρέχει την `process_action`

callback chain. Το instrumentation module εκπέμπει `start_processing.action_controller` και τυλίγει την εκτέλεση με `process_action.action_controller`. Το rescue module πιάνει exceptions, ζητά από το request να εμφανίσει detailed exceptions όπου επιτρέπεται και τα παραδίδει στο `rescue_with_handler` ή τα ξανασηκώνει.

Δεν υπάρχει ένας κρυφός central dispatcher που γνωρίζει όλα αυτά. Υπάρχει cooperative module chain. Αυτό εξηγεί δύο classes προβλημάτων. Ένα override που ξεχνά `super` κόβει τα modules από κάτω. Ένα `prepend` αλλάζει τη σειρά όλων των wrappers και μπορεί να μετακινήσει instrumentation ή rescue boundary.

```ruby
module SeatlyActionTag
  def process_action(action_name)
    Rails.event.with_context(seatly_action: action_name) { super }
  end
end

ApplicationController.prepend(SeatlyActionTag)
```

Ακόμη και αυτό το μικρό extension απαιτεί contract. Θέλουμε να τυλίγει callbacks και render; Θέλουμε context και όταν το action name είναι άγνωστο; Τι συμβαίνει αν το `super` σηκώσει exception; Το block API κάνει cleanup, αλλά η θέση στην ancestor chain αποφασίζει ποια events θα το δουν.

`performed?` δεν σταματά τη Ruby

Το Metal θεωρεί response performed όταν υπάρχει `response_body` ή όταν το response έχει committed. Αυτό επιτρέπει σε callbacks να σταματήσουν την action chain όταν έκαναν render ή redirect. Δεν κάνει non-local return από τη δική σου method. Αν καλέσεις `render` και συνεχίσεις σε επικίνδυνο write, η Ruby θα το εκτελέσει. Η προστασία από double render εμφανίζεται όταν επιχειρείται δεύτερο render, όχι αμέσως μετά το πρώτο.

Η `response_body=` μετατρέπει string σε array και τοποθετεί το body στο response. Η `set_response!` κλείνει το παλιό body αν αντικαθίσταται και αυτό υποστηρίζει resource cleanup. Η `dispatch` συνδέει request και response, καλεί `process`, κάνει commit το flash και μετατρέπει την απόκριση σε Rack tuple.

Callbacks ως compiled chain

Τα `before_action`, `around_action` και `after_action` χρησιμοποιούν Active Support callbacks με conditions. Η chain συντάσσεται και cache-άρεται ώστε το hot path να μην ξαναχτίζει όλη τη δομή σε κάθε request. Όταν αλλάζεις callbacks δυναμικά μετά το boot, ακυρώνεις αυτή τη σταθερότητα και δημιουργείς shared mutable class state. Σε threaded server αυτό είναι κακή ιδέα ακόμη αν «δουλεύει» στο development.

Το around callback πρέπει να κάνει yield ακριβώς μία φορά εκτός αν το contract λέει συνειδητά ότι μπορεί να short-circuit. Αν δεν κάνει yield, το action και τα inner callbacks δεν τρέχουν. Αν κάνει yield δύο φορές, εκτελείς use case δύο φορές. Η δύναμή του είναι control flow, όχι απλό decoration.

RUBY

```ruby
class ApplicationController < ActionController::Base
  around_action :with_seatly_context

  private

  def with_seatly_context
    Rails.event.with_context(request_id: request.request_id) do
      yield
    end
  end
end
```

Δες την πραγματική class σου

Η ακριβής chain εξαρτάται από το αν χρησιμοποιείς `ActionController::Base`, `ActionController::API` και από modules της εφαρμογής ή gems. Το source του Rails δίνει default composition, αλλά η runtime class είναι η αλήθεια. Τύπωσε ancestors και `method(:process_action).super_method` επαναληπτικά. Αυτό δείχνει ποια implementation θα πάρει τον έλεγχο και σε ποια σειρά.

SOURCE TRACE

Ξεκίνα από `action_controller/metal.rb` και τη λίστα modules στο `action_controller/base.rb`. Ακολούθησε μετά `process` και `process_action` στα `abstract_controller/base`, `callbacks`, `instrumentation` και `rescue` modules. Η σειρά των `include` είναι μέρος του trace.

ΠΑΓΙΔΑ

Ένα monkey patch στο `process_action` είναι framework integration. Έχει μεγαλύτερο blast radius από ένα concern σε έναν controller. Αν είναι αναγκαίο, κράτα το μικρό, έλεγξε method parameters και βάλε request tests για success, handled error, unhandled error και committed response.

ΜΗΧΑΝΙΣΜΟΣ

Ο controller pipeline είναι ancestor chain που συνεργάζεται μέσω `super`. Για να ξέρεις το πραγματικό boundary ενός callback ή exception, ακολούθησε τη lookup order της class που τρέχει.

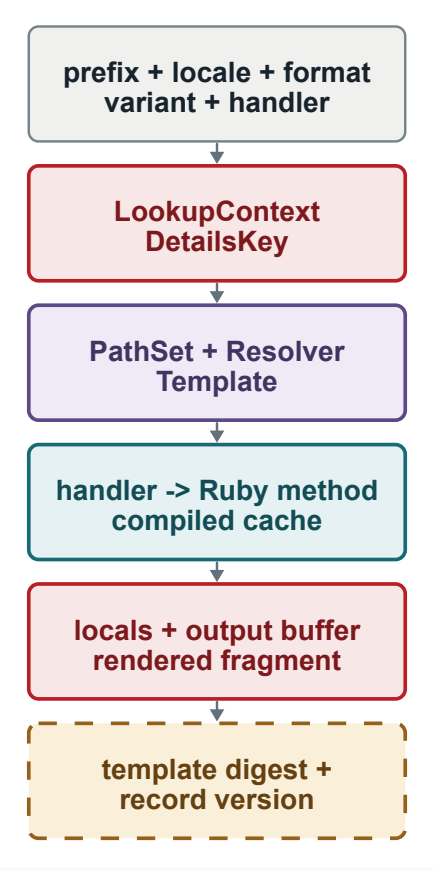
PROBE

Στην console πάρε `ApplicationController.instance_method(:process_action)` και ακολούθησε πέντε φορές το `super_method`. Τύπωσε owner και source location. Μετά κάνε request που κάνει render, redirect και raise. Συσχέτισε τα logs με τη chain που βρήκες.

ΚΕΦΑΛΑΙΟ 6

Lookup, compilation και cache των views

Ένα `render` δεν ανοίγει απλώς ένα ERB αρχείο. Επιλύει dimensions, cache keys, compiled methods, dependencies και output buffers.



Σχήμα 6.1 · Το view pipeline από details και resolver ως compiled template.

Το Action View πρέπει να απαντήσει πολύ περισσότερα από «ποιο filename;». Χρειάζεται prefixes, locale, format, variant, handler και locals. Το `ActionView::LookupContext` κρατά view paths και details για το request. Το `DetailsKey` μετατρέπει τις dimensions σε canonical key ώστε οι resolvers να χρησιμοποιούν cache χωρίς να ξαναϋπολογίζουν το ίδιο request shape.

Αυτό εξηγεί γιατί ένα αθώο `formats=` ή variant μπορεί να αλλάξει cache path. Τα registered details στην έκδοση αναφοράς είναι locale, formats, variants και handlers. Τα prefixes προκύπτουν από

controller path ή explicit template path. Ο resolver ψάχνει candidates και επιστρέφει `ActionView::Template`, όχι ήδη παραγόμενο HTML.

Από Template σε Ruby method

Το template έχει source, identifier, handler, format και metadata. Στην πρώτη εκτέλεση, ο handler μετατρέπει το source σε Ruby code. Το Action View ορίζει compiled method σε generated module ή view context class και μετά την καλεί με locals και output buffer. Η compilation cache αφαιρεί parsing και method definition από τα επόμενα renders. Σε development, reload και resolver cache invalidation πρέπει να απομακρύνουν methods που αντιστοιχούν σε παλιό source.

Η λεπτομέρεια αυτή αλλάζει το debugging. Αν ένα template exception δείχνει generated method name, δεν σημαίνει ότι χάθηκε το αρχικό location. Το template κρατά identifier και line mapping. Αν ένα monkey patch αλλάξει handler output, το effect μπορεί να μην φανεί μέχρι να καθαριστεί η compiled cache.

Locals είναι μέρος του compile shape

Τα strict locals επιτρέπουν σε template magic comment να δηλώσει signature. Ακόμη και χωρίς strict locals, το σύνολο local names μπορεί να επηρεάζει τη compiled method. Δύο renders του ίδιου partial με διαφορετικά local keys δεν είναι κατ' ανάγκη το ίδιο compiled call shape. Ένα dynamic hash με δεκάδες σπάνιους συνδυασμούς δημιουργεί περισσότερη compilation και cache πίεση από ένα σταθερό interface.

```erb
<%# locals: (reservation:, compact: false) %>
<article id="<%= dom_id(reservation) %>">
  <%= reservation.reference %>
  <% unless compact %>
    <%= reservation.status %>
  <% end %>
</article>
```

Το partial γίνεται μικρό interface. Το benefit δεν είναι μόνο error message. Μειώνει τα πιθανά shapes και κάνει dependency ορατή. Μην περνάς ολόκληρο `local_assigns` downstream για ευκολία.

Fragment cache και template digest

Το fragment caching συνδυάζει application key με template tree digest. Το digest προσπαθεί να εντοπίσει template dependencies, ώστε αλλαγή σε partial να αλλάξει το cache key. Dynamic render paths δεν είναι πάντα statically discoverable. Τότε χρειάζεται explicit dependency comment ή manual version. Το Russian doll caching στηρίζεται επιπλέον στα `cache_key_with_version` των records. Template digest και record version λύνουν διαφορετικές διαστάσεις invalidation.

Το collection rendering έχει δικό του optimized path. Μπορεί να κάνει bulk cache reads, να χωρίσει hits και misses και να render μόνο τα misses. Αν κρύψεις queries σε helpers ή partials, το optimization του render δεν εξαφανίζει N+1 database behavior. View pipeline και data loading pipeline παραμένουν ξεχωριστά.

Streaming αλλάζει τη σειρά

Με streaming template renderer, layout και template μπορούν να παράγουν chunks πριν ολοκληρωθεί όλο το render. Τα headers δεσμεύονται νωρίτερα και exceptions αργότερα δεν μπορούν να γίνουν κανονική error page. Επιπλέον, η σειρά evaluation μπορεί να κάνει queries αργότερα από ό,τι περιμένει ένα απλό controller trace. Χρειάζεσαι end-to-end observation, όχι μόνο duration του `render` call.

RUBY

```ruby
ActiveSupport::Notifications.subscribe("render_template.action_view") do |event|
  Rails.logger.info(
    template: event.payload[:identifier],
    duration_ms: event.duration
  )
end
```

Σε production μην καταγράφεις full filesystem identifiers αν αποκαλύπτουν paths που δεν χρειάζεσαι. Κανονικοποίησε σε application-relative template name και έλεγξε cardinality.

SOURCE TRACE

Η αναζήτηση αρχίζει στο `ActionView::LookupContext`, περνά από `PathSet`, `Resolver` και `Template`. Για την είσοδο του render ακολούθησε τα `ActionView::Renderer` και `TemplateRenderer`. Για digest invalidation δες το `action_view/digestor.rb`. Αυτά είναι implementation paths, όχι extension API.

ΠΑΓΙΔΑ

Μην θεωρείς κάθε cache miss πρόβλημα του store. Μπορεί να άλλαξε detail key, locals shape, template digest ή record version. Κατέγραψε τα components του key πριν αυξήσεις TTL ή αλλάξεις backend.

ΜΗΧΑΝΙΣΜΟΣ

Το render pipeline έχει δύο διαφορετικά caches: compilation και rendered content. Το πρώτο αποφεύγει να ξαναφτιάξει Ruby method. Το δεύτερο αποφεύγει να ξανατρέξει το template. Η invalidation τους έχει διαφορετικές αιτίες.

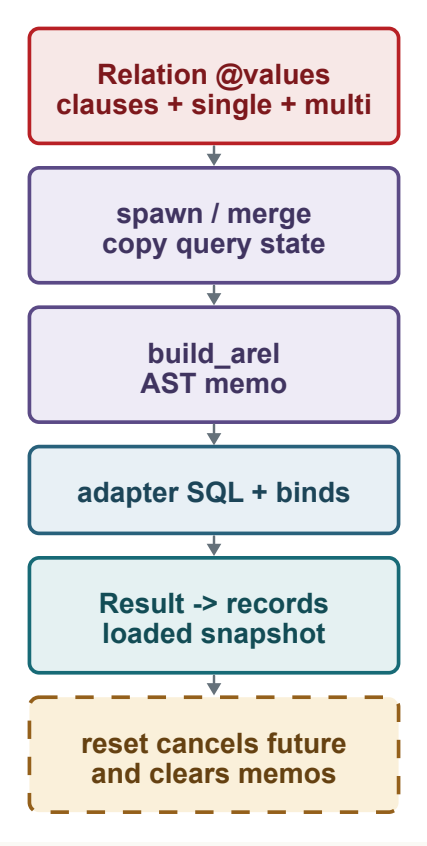
PROBE

Διάλεξε ένα cached collection partial. Μέτρησε `render_collection.action_view`, `render_partial.action_view` και cache hits σε δύο ίδια requests. Μετά άλλαξε μόνο locale, μόνο variant και μόνο ένα child partial. Κατέγραψε ποιο layer έχασε το cache και γιατί.

ΚΕΦΑΛΑΙΟ 7

Η Relation ως επίμονο query state

Μια `Relation` δεν είναι ούτε SQL string ούτε array. Είναι query state που αντιγράφεται, συντίθεται, μεταγλωττίζεται και μόνο τότε αποκτά records.



Σχήμα 7.1 · Μια Relation περνά από values σε Arel, SQL, result και records.

Η `ActiveRecord::Relation` κρατά model, table, predicate builder και ένα hash `@values`. Τα keys χωρίζονται νοητικά σε multi-value, single-value και clause values. `order`, `joins` και `select` συσσωρεύουν arrays. `limit`, `lock` και `distinct` κρατούν μία τιμή. `where` και `having` κρατούν clause objects. Αυτό το state είναι η πραγματική γλώσσα σύνθεσης πριν από το Arel.

Οι query methods συνήθως καλούν `spawn` και μετά μια bang variant. Το public `where` επιστρέφει άλλο relation. Το internal `where!` αλλάζει το αντίγραφο. Κατά το `dup`, το `initialize_copy` κάνει shallow duplicate του `@values` και καλεί `reset`, ώστε compiled Arel, SQL, loaded records και async future να μην διαρρεύσουν στο νέο branch. Δεν είναι mathematically immutable object. Είναι copy-on-write discipline που οι public methods διατηρούν.

```ruby
base = Reservation.where(event_id: 42)
held = base.where(status: :held)
ordered = base.order(created_at: :desc)

puts base.to_sql
puts held.to_sql
puts ordered.to_sql
```

Το `base` δεν αποκτά τα φίλτρα των παιδιών. Αν όμως extension πειράξει arrays μέσα στο `values` ή καλέσει private bang method, σπάει αυτή τη νοητική εγγύηση. Γι' αυτό η ασφαλής επέκταση γίνεται με scopes και public query methods.

Merge δεν είναι απλό hash merge

Το `merge` χρειάζεται semantics ανά clause. Ένα `where` με ίδιο attribute μπορεί να αντικαταστήσει predicate. Joins από διαφορετικές relations πρέπει να συνδυαστούν με reflection-aware τρόπο. `rewhere`, `unscope`, `except` και `only` τροποποιούν διαφορετικά επίπεδα state. Η σειρά των scopes επομένως μπορεί να αλλάξει SQL, όχι επειδή το Rails είναι αυθαίρετο αλλά επειδή οι operators δεν είναι όλοι commutative.

Αν θέλεις reusable policy scope, έλεγξε τις algebraic ιδιότητες που χρειάζεσαι. Ένα tenant scope πρέπει να επιβιώνει από downstream merge; Τότε μην αφήνεις caller να το αφαιρεί με γενικό `unscope`. Βάλε tenant constraint σε ασφαλέστερο boundary ή ελέγξτε explicit composition. Το relation API δεν μετατρέπει από μόνο του authorization σε invariant.

Η στιγμή του materialization

Μέχρι να ζητήσεις records, το relation είναι lazy. `each`, `to_a`, `first`, `pluck`, calculation ή inspect μπορεί να προκαλέσει query με διαφορετικό path. Το `loaded?` σημαίνει ότι το relation έχει records cache, όχι ότι κάθε association τους είναι loaded. Το `to_sql` χτίζει και memoizes SQL χωρίς να κάνει query. Το `reset` ακυρώνει `@arel`, `@to_sql`, records, cache keys και pending future.

Το subtle point είναι ότι ένα loaded relation είναι stateful snapshot. Νέο row στη database δεν εμφανίζεται σε δεύτερο `to_a` στο ίδιο instance. Το `reload` κάνει reset και load. Αν περάσεις loaded relation σε μακρόβιο object, δεν περνάς «ένα query». Περνάς cached result με identity και χρόνο ζωής.

```ruby
scope = Reservation.where(event_id: 42)
first_count = scope.to_a.length
Reservation.create!(event_id: 42, status: :held)
cached_count = scope.to_a.length
fresh_count = scope.reload.to_a.length

puts [first_count, cached_count, fresh_count].inspect
```

Instantiation είναι ξεχωριστό κόστος

Το adapter επιστρέφει `ActiveRecord::Result`. Μετά το relation κάνει instantiate records, επιλέγει STI classes όπου χρειάζεται, τρέχει instantiation hooks και preloads associations. Ένα query με μικρό

database time μπορεί να έχει μεγάλο Ruby time λόγω χιλιάδων objects και attribute sets. Το SQL event από μόνο του δεν μετρά όλο το relation materialization.

Για projections, `pluck` αποφεύγει full model instantiation. Για domain behavior, models μπορεί να είναι σωστά. Το decision χρειάζεται δύο μετρήσεις: database execution και Ruby allocation/instantiation. Μην βαφτίζεις κάθε μεγάλο result «αργό query».

SOURCE TRACE

Δες το state inventory και το lifecycle στο `active_record/relation.rb`, τη σύνθεση στο `relation/query_methods.rb` και τα `spawn`, `merge`, `except`, `only` στο `relation/spawn_methods.rb`. Ακολούθησε `where -> spawn -> where! -> build_arel -> exec_main_query`.

ΠΑΓΙΔΑ

Μην memoize-άρεις loaded relations σε singleton, class variable ή process cache. Κρατούν records, ίσως stale associations και objects από συγκεκριμένο execution. Memoize query parameters ή IDs και ξαναχτίσε relation στο σωστό context.

ΜΗΧΑΝΙΣΜΟΣ

Η Relation έχει δύο ζωές: query program πριν από το load και cached record set μετά. Πολλά bugs προκύπτουν όταν ο caller νομίζει ότι κρατά την πρώτη ενώ έχει ήδη περάσει στη δεύτερη.

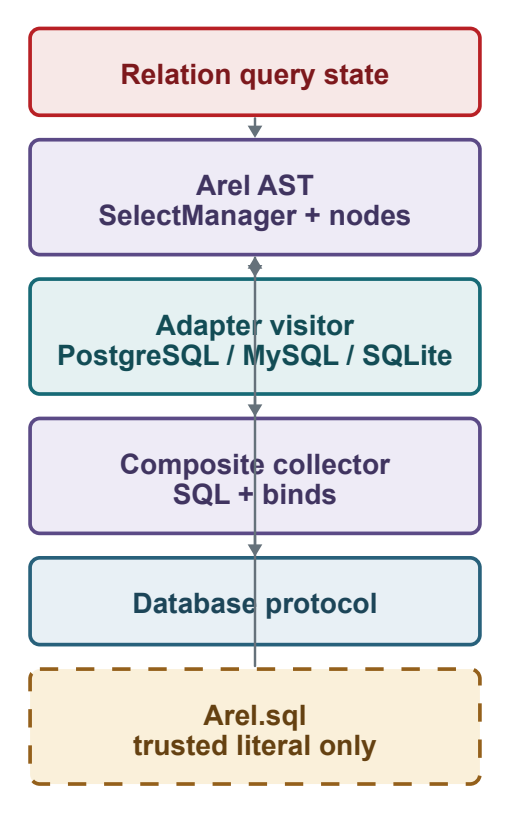
PROBE

Πάρε ένα production-like scope και τύπωσε `values`, `to_sql`, `loaded?` πριν και μετά από `each`. Μέτρησε SQL duration και συνολικό monotonic duration. Σύγκρινε full records με `pluck`. Αν η διαφορά βρίσκεται μετά το SQL event, έχεις instantiation ή preload κόστος, όχι planner πρόβλημα.

ΚΕΦΑΛΑΙΟ 8

Arel visitors, collectors και binds

Το Arel δεν «φτιάχνει SQL» σε ένα βήμα. Χτίζει AST και ο adapter-specific visitor το περπατά γράφοντας SQL και binds σε collectors.



Σχήμα 8.1 · Το query state μετατρέπεται σε AST και μετά σε adapter-specific output.

Το `build_arel` μετατρέπει το state της Relation σε `Arel::SelectManager`. Predicates, joins, projections, grouping, ordering, limits, common table expressions και locks γίνονται nodes. Αυτό κρατά τη δομή του query ξεχωριστή από το SQL dialect. Η PostgreSQL, η MySQL και η SQLite μπορούν να επισκεφθούν το ίδιο γενικό AST με διαφορετικές rules.

Ο visitor εφαρμόζει double dispatch. Ένα node καλεί ουσιαστικά method της μορφής `visit_Arel_Nodes_Equality`. Η method γράφει tokens σε collector και επισκέπτεται children. Το αποτέλεσμα δεν χρειάζεται να είναι απλό string. Composite collector μπορεί να συλλέγει ταυτόχρονα SQL και bind values. Άλλος collector αντικαθιστά binds για logging. Αυτή η διάκριση είναι ο λόγος που το ασφαλές query δεν χρειάζεται να interpolates user values μέσα στο SQL text.

QueryAttribute, όχι γυμνή τιμή

Όταν το predicate builder βλέπει `status: :held`, συνδέει Arel attribute με `ActiveRecord::Relation::QueryAttribute`. Το query attribute γνωρίζει name, value και type. Μπορεί να αποφασίσει database serialization πριν ο adapter στείλει bind. Το SQL shape μένει σταθερό ενώ οι values αλλάζουν, κάτι που επιτρέπει prepared statements και ασφαλές quoting.

```ruby
relation = Reservation.where(event_id: 42, status: :held)
puts relation.to_sql

relation.bound_attributes if relation.respond_to?(:bound_attributes, true)
```

Η δεύτερη γραμμή είναι σκόπιμα guarded επειδή η πρόσβαση στα internal binds δεν είναι public contract. Για application debugging προτίμησε `sql.active_record` payloads, με προσοχή στο αν επιτρέπεται να καταγράψεις values. Το `to_sql` είναι χρήσιμο για shape, αλλά δεν αποδεικνύει ακριβώς το wire protocol ή το prepared statement name που χρησιμοποίησε ο adapter.

Γιατί υπάρχουν adapter visitors

Το SQL standard δεν εξαφανίζει τις διαφορές. Quoting identifiers, boolean literals, bind placeholders, `LIMIT`, locks, JSON operators, `RETURNING` και upsert syntax αλλάζουν. Ο connection adapter διαλέγει visitor που επεκτείνει το γενικό `ToSql`. Ένας node που υποστηρίζεται σε PostgreSQL μπορεί να μην έχει ισοδύναμο σε SQLite ή να μεταγλωττίζεται διαφορετικά.

Αυτό είναι boundary για portability. Ένα scope που χρησιμοποιεί μόνο public Active Record predicates έχει μεγαλύτερη πιθανότητα να περάσει adapters. Ένα `Arel.sql` δηλώνει ότι αναλαμβάνεις την ασφάλεια και τη dialect dependence του fragment. Δεν είναι sanitizer. Είναι marker ότι το string θεωρείται known SQL.

```ruby
safe_order = Arel.sql("held_until ASC NULLS LAST")
reservations = Reservation.order(safe_order)

puts reservations.to_sql
```

Η literal εδώ είναι σταθερή και ελεγχόμενη από source code. Αν περιέχει params, χρησιμοποίησε public APIs με binds ή `sanitize_sql_array` όπου είναι το σωστό contract. Μην τυλίξεις user input σε `Arel.sql` για να περάσει ένα safety check.

AST transformations και correctness

Scopes που συνδυάζουν joins μπορούν να δημιουργήσουν duplicate rows. Η AST είναι syntactically έγκυρη αλλά το result semantics λάθος. Ένα visitor δεν γνωρίζει domain cardinality. Ούτε το Arel επιλέγει index. Μετατρέπει structure σε dialect. Correctness και plan quality παραμένουν δική σου ευθύνη.

Για δύσκολο query, τύπωσε AST class tree μόνο για relevant nodes. Παρατήρησε ποιο predicate έγινε equality, range, `IN` ή raw literal. Μετά κοίτα SQL και bind types. Αν η database κάνει implicit cast στη indexed column, το πρόβλημα μπορεί να ξεκινά από custom type serialization, όχι από τον planner.

Prepared shape και statement lifecycle

Δύο queries με ίδιο SQL shape και διαφορετικούς binds μπορούν να μοιραστούν prepared statement. Αλλαγή στο πλήθος ενός `IN` list ή dynamic projection μπορεί να παράγει άλλο shape. Το Arel output επομένως επηρεάζει το cardinality του statement pool. Μεγάλη ποικιλία dynamic SQL αυξάνει parse και cache churn, ακόμη αν κάθε μεμονωμένο query είναι γρήγορο.

SOURCE TRACE

Η μετάβαση Relation προς AST βρίσκεται στο `relation/query_methods.rb`. Ο γενικός visitor είναι το `arel/visitors/to_sql.rb` και οι adapter overrides είναι δίπλα του. Δες επίσης `arel/collectors/` και `relation/query_attribute.rb`. Τα ονόματα των nodes είναι implementation detail.

ΠΑΓΙΔΑ

Το `Arel.sql` αφαιρεί προστασία, δεν προσθέτει escaping. Χρησιμοποίησέ το μόνο για literal SQL που ελέγχει ο κώδικας και γράψε adapter-specific test όταν το fragment δεν είναι portable.

ΜΗΧΑΝΙΣΜΟΣ

Το Arel κρατά query structure. Ο visitor κρατά dialect. Ο collector κρατά το παραγόμενο SQL και τα binds. Η database κρατά τελικά το plan και τα semantics εκτέλεσης.

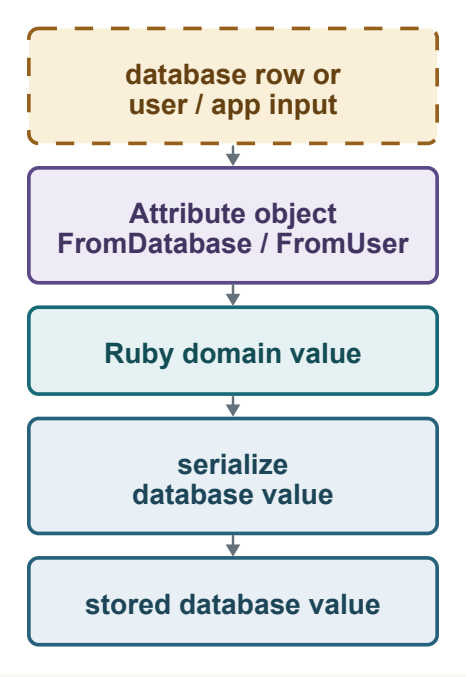
PROBE

Διάλεξε query με custom type, array predicate ή JSON operator. Κατέγραψε SQL shape, bind Ruby class, serialized database value και adapter. Τρέξε `EXPLAIN` με representative values. Αν αλλάξεις type ή adapter, επανάλαβε όλο το trace, όχι μόνο το `to_sql`.

ΚΕΦΑΛΑΙΟ 9

Attributes, types και dirty tracking

Ένα model attribute είναι μικρό state machine ανάμεσα σε raw database value, Ruby value, user assignment και serialized write.



Σχήμα 9.1 · Οι διαφορετικές προελεύσεις ενός attribute και οι lazy μετατροπές του.

Το `reservation.held_until` μοιάζει με instance variable. Στην πραγματικότητα το record κρατά `AttributeSet`, έναν χάρτη από names σε `ActiveModel::Attribute` objects. Κάθε attribute γνωρίζει type, value before type cast, original attribute και memoized cast ή serialized values. Η προέλευση έχει σημασία. Value από database χρειάζεται `deserialize`. Value που έδωσε ο χρήστης χρειάζεται `cast` και μετά `serialize`. Value που έχει ήδη γίνει cast δεν πρέπει να περάσει ξανά από τα ίδια βήματα.

Οι private subclasses `FromDatabase`, `FromUser` και `WithCastValue` εκφράζουν αυτές τις διαδρομές. Δεν τις καλούμε. Μας βοηθούν όμως να καταλάβουμε γιατί το type system είναι lazy. Το Rails μπορεί να κρατήσει raw string από result set μέχρι να διαβαστεί το attribute. Σε μεγάλο projection, columns που δεν αγγίζεις δεν χρειάζεται να deserialized αμέσως.

Το public custom type contract

Ένα custom type πρέπει να ορίζει καθαρά Ruby domain, accepted input και database representation. Το `cast` χειρίζεται application assignments. Το `deserialize` χειρίζεται values από adapter. Το

`serialize` δίνει value κατάλληλο για τον underlying column type. Συμμετρία δεν σημαίνει ότι οι τρεις methods είναι ίδιες.

```ruby
class ReservationCodeType < ActiveRecord::Type::String
  def cast(value)
    super&.strip&.upcase
  end

  def serialize(value)
    cast(value)
  end
end

ActiveRecord::Type.register(:reservation_code, ReservationCodeType)
```

Το type δεν πρέπει να κάνει query, network I/O ή να εξαρτάται από mutable request context. Καλείται σε binds, dirty checks, serialization και model reads. Ένα «έξυπνο» type μπορεί να βάλει κρυφό I/O σε planner-critical hot path.

Dirty tracking έχει δύο χρονικούς ορίζοντες

Πριν από save, methods όπως `will_save_change_to_status?` κοιτούν mutations σε σχέση με database snapshot. Μετά το save, `saved_change_to_status?` κοιτά το change που μόλις γράφτηκε. Το Active Record μετακινεί mutation trackers στο persistence boundary. Αν ένα callback χρησιμοποιήσει method από λάθος χρονικό ορίζοντα, μπορεί να πάρει λογικά σωστό αλλά άχρηστο αποτέλεσμα.

Mutable Ruby objects είναι δυσκολότερα. Αν πάρεις array ή JSON hash και το αλλάξεις in place, δεν υπάρχει setter call για να σημάνει assignment. Ο type πρέπει να μπορεί να εντοπίσει `changed_in_place?`, συχνά συγκρίνοντας serialized representation. Για δικό σου mutable value object, είτε υλοποίησε σωστά αυτή τη method είτε προτίμησε immutable object και assignment νέας τιμής.

```ruby
reservation.metadata["source"] = "partner"
puts reservation.metadata_changed?

reservation.metadata = reservation.metadata.merge("channel" => "api")
puts reservation.metadata_changed?
```

Το πρώτο μπορεί να εξαρτάται από τη συμπεριφορά του type. Το δεύτερο κάνει explicit assignment και είναι πιο καθαρό application signal.

Partial selects και missing attributes

`Reservation.select(:id, :status).first` έχει AttributeSet με uninitialized ή missing columns. Πρόσβαση σε column που δεν φορτώθηκε μπορεί να σηκώσει `ActiveModel::MissingAttributeError`. Αυτό προστατεύει από το να μπερδέψεις «δεν επιλέχθηκε» με `nil`. Το id έχει ειδική συμπεριφορά, αλλά μην βασίζεσαι σε partial record για γενικό domain update. Callback μπορεί να διαβάσει column που δεν υπάρχει στο projection.

Encryption και serialization προσθέτουν wrappers στη type chain. Το logical Ruby value, το ciphertext database value και deterministic query representation δεν είναι το ίδιο. Όταν debug-άρεις bind, μην

τυπώνεις sensitive serialized value. Κατέγραψε type class, column SQL type και presence, όχι secret content.

Equality, defaults και mutable values

Η dirty detection δεν συγκρίνει απλώς δύο Ruby objects με έναν καθολικό τρόπο. Το type μπορεί να ορίσει πώς εντοπίζεται αλλαγή από user input ή από in-place mutation του deserialized value. Ένα custom type που επιστρέφει mutable hash αλλά δεν υλοποιεί σωστά αυτό το contract μπορεί να χάσει update ή να γράφει σε κάθε save. Το bug συχνά φαίνεται μόνο μετά από reload, επειδή τότε αλλάζει η αφετηρία από `FromUser` σε `FromDatabase`.

Τα defaults είναι επίσης typed attributes. Default proc εκτελείται για κάθε instance, αλλά το αποτέλεσμα εξακολουθεί να περνά από type casting και dirty state. Απόφυγε shared mutable default object. Στο test του type έλεγξε δύο νέα records, mutation χωρίς assignment, serialize-deserialize round trip και equality μετά από reload. Μην αρκεστείς στο ότι το form εμφανίζει τη σωστή τιμή. Το persistence contract κρίνεται από το database representation που παράγεται και από το αν το framework γνωρίζει ότι πρέπει να το γράψει.

SOURCE TRACE

Ο πυρήνας βρίσκεται στα `activemodel/lib/active_model/attribute.rb` και `attribute_set.rb`. Για το public type API δες `active_record/type`, ενώ το persistence dirty boundary βρίσκεται στο `active_record/attribute_methods/dirty.rb`.

ΠΑΓΙΔΑ

Μην θεωρείς `value_before_type_cast` ασφαλές για logs. Είναι συχνά ακριβώς το μη έμπιστο ή sensitive input. Για debugging κατέγραψε class, length, nullability και validation outcome, όχι ολόκληρη τιμή.

ΜΗΧΑΝΙΣΜΟΣ

Το attribute type είναι boundary ανάμεσα σε τρεις γλώσσες: input, Ruby domain και database representation. Dirty tracking είναι σωστό μόνο αν το type δηλώνει σωστά πότε δύο representations διαφέρουν.

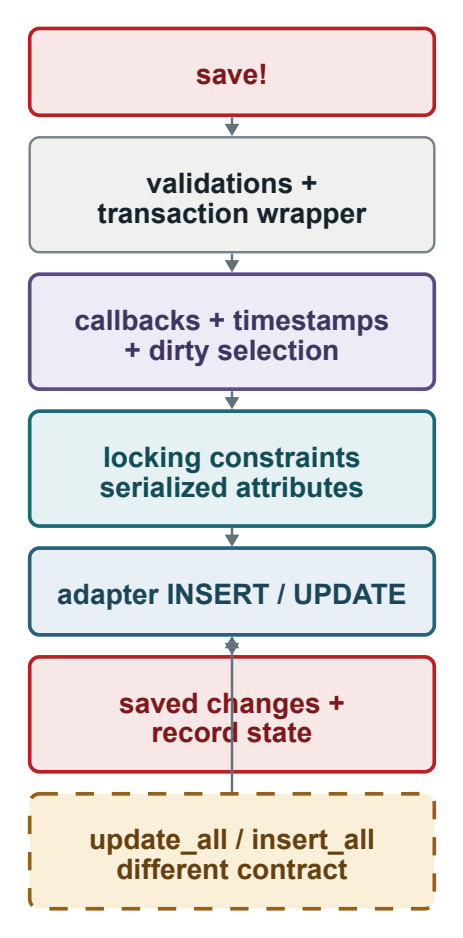
PROBE

Για ένα custom ή serialized attribute, κατέγραψε χωρίς sensitive values: `type_for_attribute`, SQL column type, `value.class`, dirty state πριν το save και saved changes μετά. Δοκίμασε assignment, in-place mutation, reload και round trip από database. Βάλε αυτά τα cases σε type unit test.

ΚΕΦΑΛΑΙΟ 10

Persistence, partial writes και callbacks

Το `save!` είναι module chain που προσθέτει validation, transaction, callbacks, timestamps, dirty selection, locking και τελικά adapter write.



Σχήμα 10.1 · Η αλυσίδα του save από validation ως adapter και state reset.

Όπως στον controller, η persistence συμπεριφορά σχηματίζεται από Ruby ancestors. Το public `save!` περνά από validations και transaction wrapper πριν φτάσει στο persistence layer. Τα `_create_record` και `_update_record` τυλίγονται από callbacks, timestamps, dirty tracking, optimistic locking, counter caches και άλλα included modules. Η ακριβής chain εξαρτάται από το model configuration.

Αυτό σημαίνει ότι το «ένα UPDATE» έχει δύο κόστη. Το database statement και όλο το Ruby work πριν και μετά. Validation μπορεί να κάνει queries. Callback μπορεί να φορτώσει associations. Autosave

μπορεί να γράψει graph. Ένα SQL trace που κοιτά μόνο το τελευταίο UPDATE χάνει το application pipeline.

Partial writes

Με `partial_updates` ενεργό, το dirty layer δίνει στην `_update_record` μόνο attributes που άλλαξαν. Με `partial_inserts`, ένα INSERT μπορεί να παραλείψει unchanged columns και να αφήσει database defaults να εφαρμοστούν. Αυτό μειώνει SQL size και αποφεύγει άσχετα writes, αλλά κάνει το dirty model μέρος του correctness.

Το persistence layer φιλτράρει alias, readonly και virtual attributes μέσω `attributes_for_update` ή `attributes_for_create`. Μετά ο adapter παίρνει serialized values και constraints. Στα updates, primary key και optimistic locking column μπορούν να μπουν στο `WHERE`, ώστε zero affected rows να σηματοδοτήσει stale object αντί να χαθεί update.

```ruby
reservation = Reservation.find(42)
reservation.status = :confirmed
reservation.save!

puts reservation.saved_changes.inspect
```

Μετά το successful write, το record ενημερώνει `@new_record`, previous changes και dirty trackers. Με rollback, το transaction layer προσπαθεί να επαναφέρει record state. Αυτό δεν αναιρεί arbitrary side effects που έκανε callback σε εξωτερικό σύστημα.

Callback chain και transaction boundary

Το `save` τυλίγεται σε transaction ακόμη όταν δεν άνοιξες explicit block, ώστε validation, callbacks και write να έχουν ενιαίο database boundary. Τα ordinary `after_save` και `after_create` τρέχουν πριν από commit. Μόνο `after_commit` σημαίνει ότι η outermost relevant transaction ολοκληρώθηκε. Αν ένα nested save μπει σε existing transaction, το callback πρέπει να περιμένει την outer commit.

Η σειρά callbacks είναι compiled class state. Conditional callbacks μπορούν να διαβάσουν dirty state, αλλά η επιλογή της σωστής method εξαρτάται από το αν βρίσκεσαι before ή after persistence. Ένα callback που κάνει δεύτερο `save!` στο ίδιο record μπορεί να αλλάξει mutation trackers και να κάνει το trace πολύ πιο δύσκολο. Προτίμησε να υπολογίσεις όλες τις attribute αλλαγές πριν από το write.

Τα bypass APIs είναι διαφορετικό contract

`update_columns` γράφει απευθείας attributes χωρίς validations και callbacks, αν και περνά type serialization και καθαρίζει dirty flags για τα attributes. `update_all` λειτουργεί σε relation και δεν instantiates records. `insert_all` και `upsert_all` έχουν bulk contract. Δεν είναι «γρηγορότερα save». Είναι άλλος μηχανισμός με άλλες εγγυήσεις.

Αν domain invariant υλοποιείται μόνο σε callback, ένα bulk writer το παρακάμπτει. Αν ο invariant πρέπει να ισχύει για κάθε writer, βάλε database constraint. Για derived application behavior, κάνε explicit bulk operation που ενημερώνει ό,τι χρειάζεται και έχει δικά της tests.

RUBY

```ruby
updated = Reservation.where(status: :held)
  .where(held_until: ...Time.current)
  .update_all(status: Reservation.statuses.fetch(:expired))

Rails.event.notify("seatly.holds.expired", count: updated)
```

Εδώ η operation δηλώνει συνειδητά ότι δεν περιμένει per-record callbacks. Η event emission είναι aggregate και πρέπει να γίνει μετά από successful statement ή commit, ανάλογα με το surrounding transaction.

In-memory success δεν είναι committed success

Μετά το SQL, το model μπορεί να έχει νέο id, timestamps, cleared dirty state και αυξημένο `lock_version`, ενώ η outer transaction μπορεί ακόμη να αποτύχει. Το Rails κρατά transaction state ώστε να αποκαταστήσει κρίσιμα persistence fields σε rollback, αλλά δεν μπορεί να αναιρέσει κάθε αυθαίρετη mutation που έκανε callback σε άλλο Ruby object. Αυτός είναι ένας ακόμη λόγος να μην κρύβεις domain workflow μέσα σε model callbacks.

Το `after_save` βλέπει επιτυχημένο statement. Το `after_commit` βλέπει durable database outcome. Κανένα από τα δύο δεν αποδεικνύει ότι εξωτερικό API έλαβε ένα effect. Όταν διαβάζεις bug report με «το record σώθηκε αλλά το email χάθηκε», το πρώτο βήμα είναι να σημειώσεις ακριβώς σε ποιο από αυτά τα τρία boundaries βρισκόταν κάθε side effect. Μετά επιλέγεις callback, outbox ή idempotent worker με βάση την απαιτούμενη εγγύηση.

SOURCE TRACE

Ακολούθησε `save!` στα `validations.rb`, `transactions.rb` και `persistence.rb`. Οι wrappers `_create_record` και `_update_record` βρίσκονται στα `callbacks.rb`, `timestamp.rb`, `attribute_methods/dirty.rb` και `locking/optimistic.rb`. Χρησιμοποίησε runtime `super_method` για την ακριβή model chain.

ΠΑΓΙΔΑ

Μην αντικαθιστάς `save!` με `update_all` για performance χωρίς να καταγράψεις ποια validations, callbacks, timestamp updates, locking checks και side effects χάνονται. Η διαφορά είναι correctness contract, όχι μόνο ταχύτητα.

ΜΗΧΑΝΙΣΜΟΣ

Το persistence pipeline αποφασίζει τρία πράγματα: αν επιτρέπεται το write, ποια columns θα γραφτούν και ποιο in-memory state θα μείνει μετά. Η database statement είναι μόνο η στενή μέση του pipeline.

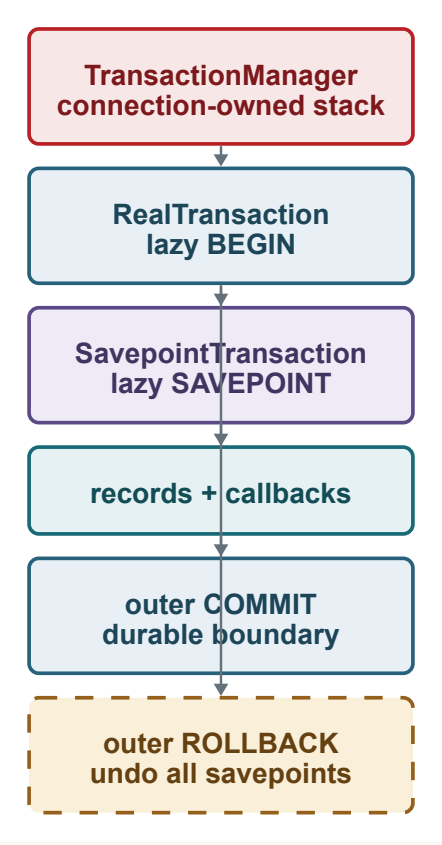
PROBE

Για ένα αργό model save, ακολούθησε `method(:save!).super_method` και μέτρησε notifications για SQL μαζί με συνολικό monotonic χρόνο. Τύπωσε μόνο τα names από `changes_to_save`. Δοκίμασε stale `lock_version` και rollback. Επιβεβαίωσε ποιο callback τρέχει και ποιο state επανέρχεται.

ΚΕΦΑΛΑΙΟ 11

TransactionManager, savepoints και commit records

Ένα transaction block δεν αντιστοιχεί πάντα σε άμεσο `BEGIN`. Το Rails κρατά stack από transaction objects, κάνει lazy materialization και μεταφέρει commit callbacks προς το πραγματικό durable boundary.



Σχήμα 11.1 · Η stack από RealTransaction και SavepointTransaction ως το outer commit.

Κάθε connection έχει `TransactionManager`. Η manager κρατά stack με `RealTransaction`, `SavepointTransaction` ή `RestartParentTransaction`. Έξω από transaction εκθέτει `NullTransaction`. Όταν ανοίγει block, επιλέγει object με βάση `requires_new`, joinability, isolation και adapter capabilities. Αυτό το Ruby object υπάρχει πριν χρειαστεί απαραίτητα database `BEGIN`.

Lazy materialization

Αν ένα transaction block δεν κάνει database operation, το Rails μπορεί να μη στείλει ποτέ `BEGIN`. Η `RealTransaction#materialize!` καλεί το adapter για `begin_db_transaction` ή isolated begin και ξεκινά transaction instrumentation. Nested savepoint materialize-άρεται μόνο όταν χρειαστεί, με `create_savepoint`. Αυτή η βελτιστοποίηση σημαίνει ότι το χρονικό σημείο εισόδου στο Ruby block δεν είναι υποχρεωτικά το wire-level start.

Το distinction φαίνεται σε instrumentation. `start_transaction.active_record` και `transaction.active_record` σχετίζονται με materialized transaction. Αν μετράς πόσα Ruby blocks άνοιξαν, χρειάζεσαι application instrumentation. Αν μετράς database transaction duration, χρησιμοποίησε τα Active Record events.

Join, savepoint και durability

Ένα nested `transaction` χωρίς `requires_new` συνήθως joins την υπάρχουσα transaction. Το `ActiveRecord::Rollback` μπορεί να καταναλωθεί από το inner block χωρίς να αναγκάσει outer rollback με τον τρόπο που περιμένει ένας caller. Με `requires_new: true`, το Rails δημιουργεί savepoint όταν ο adapter το υποστηρίζει. Release του savepoint δεν είναι durable commit. Η outer transaction μπορεί ακόμη να κάνει rollback τα πάντα.

```ruby
Reservation.transaction do
  reservation = Reservation.lock.find(42)
  reservation.update!(status: :confirmed)

  Reservation.transaction(requires_new: true) do
    AuditEntry.create!(reservation_id: reservation.id, action: "confirm")
  end

  raise ActiveRecord::Rollback if reservation.payment_missing?
end
```

Το audit row δεν επιβιώνει αν η outer transaction κάνει rollback. Αν χρειάζεσαι ανεξάρτητη durability, θέλεις άλλο connection και συνειδητό distributed boundary, όχι απλώς savepoint.

Records συμμετέχουν στην transaction

Όταν record γράφεται, προστίθεται στην current transaction. Η transaction κρατά records ώστε στο τέλος να καλέσει `committed!` ή `rolledback!`. Στην commit path επιλέγει ποιο Ruby instance θα πάρει transactional callbacks όταν πολλαπλά instances αναπαριστούν την ίδια database row. Η identity εδώ βασίζεται σε record equality και object identity rules, όχι σε έναν global identity map.

Αυτό δημιουργεί subtle failure mode. Αν φορτώσεις την ίδια row δύο φορές και αλλάξεις διαφορετικά instances μέσα στην ίδια transaction, μόνο συγκεκριμένο instance μπορεί να πάρει callback με state που το Rails θεωρεί αντιπροσωπευτικό. Μην βασίζεις κρίσιμο side effect σε ποιο object copy κέρδισε. Κράτα μία instance ανά aggregate operation ή καταχώρισε explicit transaction callback με stable ID.

Στις nested transactions που δεν τρέχουν τελικά callbacks, records και callback objects προωθούνται προς την parent. Μόνο το outer boundary τρέχει πραγματικό `after_commit`. Αυτό είναι ο μηχανισμός πίσω από enqueue-after-commit και outbox publication hooks.

Transaction callback ως local coordination

Η public transaction object μπορεί να δεχτεί after commit callback. Είναι χρήσιμο για process-local follow-up που δεν πρέπει να γίνει σε rollback. Δεν είναι durable queue. Αν η process πεθάνει αφού η database κάνει commit αλλά πριν τρέξει callback, το effect χάνεται. Για απαραίτητο external effect, γράψε outbox row μέσα στην transaction.

```ruby
Reservation.transaction do |transaction|
  reservation = Reservation.create!(event_id: 42, status: :held)
  transaction.after_commit do
    Rails.event.notify("seatly.reservation.committed", id: reservation.id)
  end
end
```

Το event εδώ είναι observability hint. Δεν χρησιμοποιείται ως μοναδική εντολή χρέωσης ή αποστολής εισιτηρίου.

Restart, savepoint και Ruby state

Ο adapter μπορεί σε ορισμένες περιπτώσεις να κάνει restart μια parent transaction αντί να δημιουργήσει κανονικό nested savepoint. Αυτό είναι implementation choice με adapter constraints, όχι application guarantee. Το public contract που μπορείς να χρησιμοποιήσεις είναι αν το block συμμετέχει στην parent ή έχει `requires_new` boundary. Το ακριβές SQL πρέπει να το δεις στο adapter που τρέχεις.

Ακόμη και όταν ένα savepoint επαναφέρει τέλεια τη database, δεν γυρίζει πίσω τον υπόλοιπο Ruby κόσμο. Array pushes, memoized values και calls σε εξωτερικές υπηρεσίες μένουν. Τα Active Record instances παίρνουν ειδική transaction bookkeeping, αλλά δεν είναι γενικό software transactional memory. Ένα retryable transaction block πρέπει συνεπώς να υπολογίζει ξανά inputs, να αποφεύγει process-global mutation και να βγάζει τα external effects έξω από το retry window. Αλλιώς η δεύτερη εκτέλεση δεν είναι ισοδύναμη με την πρώτη.

Callback records και object identity

Η transaction δεν κρατά απλώς blocks. Συλλέγει records που χρειάζονται `committed!` ή `rolledback!` και στο τέλος αποφασίζει ποια instances θα πάρουν callbacks. Αν έχεις φορτώσει την ίδια database row σε δύο διαφορετικά Ruby objects, μην υποθέτεις ότι και τα δύο θα εκτελέσουν ισοδύναμο `after_commit`. Η persistence identity είναι το row, ενώ η Ruby identity είναι το object. Η ασυμφωνία κάνει callback-driven side effects δύσκολα να διαβαστούν.

Κράτα το commit callback κοντά στο write που δημιουργεί το intent και πέρασε stable ids, όχι mutable model object, στο επόμενο layer. Αν χρειάζεται να ξαναδιαβαστεί state μετά το commit, κάν' το συνειδητά. Το instance που έχεις στο closure μπορεί να έχει in-memory mutations που δεν γράφτηκαν ή association cache από πριν το commit.

Η callback σειρά είναι επίσης μέρος του local behavior, όχι distributed ordering guarantee. Δύο processes που commit-άρουν ανεξάρτητες transactions δεν αποκτούν global σειρά επειδή και τα δύο έχουν `after_commit`. Αν downstream consumer απαιτεί order, γράψε aggregate sequence μέσα στην ίδια transaction και κάνε τον consumer να ελέγχει expected version. Το callback απλώς δηλώνει ότι το local commit ολοκληρώθηκε.

SOURCE TRACE

Η stack και τα states βρίσκονται στο `connection_adapters/abstract/transaction.rb`. Ακολούθησε `TransactionManager#within_new_transaction`, `materialize_transactions`, `commit_transaction` και `rollback_transaction`. Η συμμετοχή records και η επαναφορά in-memory state βρίσκονται στο `active_record/transactions.rb`.

ΠΑΓΙΔΑ

`after_commit` κλείνει το database rollback window. Δεν κλείνει το process crash window. Αν το επόμενο βήμα πρέπει οπωσδήποτε να συμβεί, χρειάζεσαι durable record και reconciler, όχι μόνο callback.

ΜΗΧΑΝΙΣΜΟΣ

Η nested transaction είναι control-flow και savepoint mechanism. Η durability ανήκει στο outer database commit. Όλα τα commit-dependent effects πρέπει να ευθυγραμμίζονται με εκείνο το boundary.

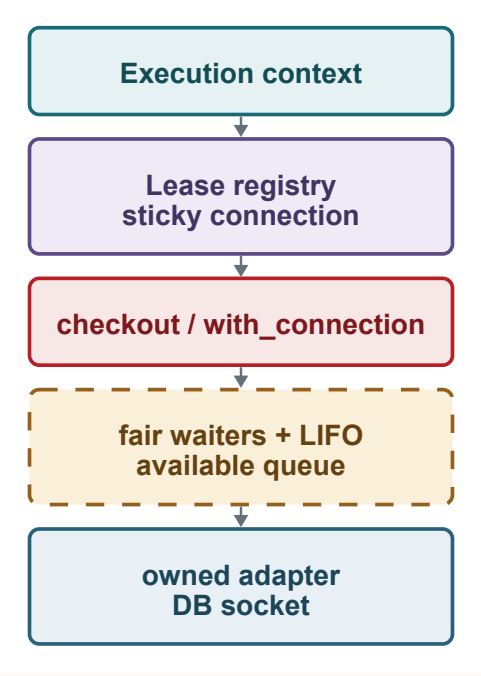
PROBE

Κάνε subscribe στα `start_transaction.active_record` και `transaction.active_record`. Τρέξε empty block, read-only block, nested joined block και `requires_new` block. Κατέγραψε SQL `BEGIN`, `SAVEPOINT`, `RELEASE` και outcome. Μετά βάλε outer rollback και επιβεβαίωσε ποια callbacks εκτελούνται.

ΚΕΦΑΛΑΙΟ 12

Connection leases, pinning και checkout queues

Το pool δεν μοιράζει απλώς sockets. Διαχειρίζεται ownership, leases, waiters, lazy connection creation, maintenance και cleanup ανά execution context.



Σχήμα 12.1 · Η διαδρομή από execution context σε lease, queue και database connection.

Ένα `ConnectionPool` ανήκει σε συγκεκριμένο connection configuration, role και shard. Κρατά τα connections που δημιούργησε, μια queue διαθέσιμων connections, leases για execution contexts, counts για concurrent connection creation και maintenance, reaper, schema cache και ίσως async query executor. Το `pool: 10` είναι μόνο ένα από αυτά τα constraints.

Lease και πραγματικό ownership

Το `lease_connection` βρίσκει το lease του current context. Αν δεν έχει connection, κάνει checkout. Μετά το μαρκάρει sticky. Το lease registry είναι γρήγορο cache. Η authoritative ownership βρίσκεται πάνω στο connection object. Αυτή η διάκριση επιτρέπει στο pool να εντοπίζει in-use connections, να κλέβει connection από dead thread κατά το reap και να μην εξαρτά correctness μόνο από ένα thread-to-connection map.

Το `with_connection` προτιμά υπάρχον leased connection. Αν δεν υπάρχει, κάνει checkout, yield και checkin σε ensure, εκτός αν το lease έγινε permanent μέσα στο block. Το public application code πρέπει σχεδόν πάντα να αφήνει το Active Record να χρησιμοποιήσει αυτό το bounded lifecycle. Ένα raw `checkout` μεταφέρει σε εσένα την ευθύνη για `checkin` σε κάθε path.

```ruby
RUBY

pool = ActiveRecord::Base.connection_pool

pool.with_connection do |connection|
  connection.select_value("SELECT 1")
end

puts pool.stat.inspect
```

Η `stat` δίνει `size`, `connections`, `busy`, `dead`, `idle`, `waiting` και `checkout_timeout` στην τρέχουσα έκδοση. Είναι snapshot, όχι histogram. Για capacity χρειάζεσαι sampling ή instrumentation της checkout αναμονής.

Η queue είναι LIFO με fair waiters

Η internal queue προτιμά πρόσφατα returned connections για locality, ενώ η condition-variable αναμονή προσπαθεί να εξυπηρετεί waiters δίκαια. Αν δεν υπάρχει available connection και το pool μπορεί ακόμη να μεγαλώσει, άλλο path δημιουργεί connection έξω από το μεγάλο synchronized section. Το `@now_connecting` αποτρέπει να ξεπεραστεί το max από ταυτόχρονους creators.

Όταν το pool φτάσει capacity, ο caller περιμένει ως `checkout_timeout`. Το `ConnectionTimeoutError` δεν αποδεικνύει ότι η database είναι αργή. Μπορεί να σημαίνει leaked lease, μακριά transaction, external I/O ενώ κρατιέται connection, περισσότερα web threads από connections ή async queries που καταναλώνουν το ίδιο budget.

Pinning είναι ειδικός μηχανισμός

Το pinning κρατά ένα connection και ανοίγει μη joinable transaction. Χρησιμοποιείται κυρίως από transactional tests ώστε διαφορετικά access paths να βλέπουν το ίδιο test transaction. Υποστηρίζει nesting depth και adapter-specific lock behavior. Δεν είναι general application API για «να είμαστε σίγουροι ότι έχουμε το ίδιο connection».

Με multi-threaded system tests, pinning και connection sharing έχουν δύσκολες συνέπειες. Άλλο thread μπορεί να χρειάζεται independent connection, ενώ το test περιμένει uncommitted rows να είναι ορατές. Όταν το behavior εξαρτάται από πραγματικό commit, transactional test είναι λάθος environment για το συγκεκριμένο case.

Reaper, idle flush και process lifecycle

Ο reaper ανακτά connections των οποίων ο owner thread πέθανε. Αν connection είναι ακόμη active, το reset και checkin το επιστρέφουν. Διαφορετικά αφαιρείται. Αυτό είναι recovery, όχι υποκατάστατο σωστού ensure. Τα idle connections μπορούν να κλείσουν με βάση `idle_timeout`, ενώ `min_connections` και prepopulation επηρεάζουν πόσο χαμηλά πέφτει το pool.

Μετά από fork, inherited database sockets δεν πρέπει να χρησιμοποιηθούν από parent και child σαν να ήταν ανεξάρτητα. Ο server lifecycle πρέπει να αποσυνδέει ή να επανεγκαθιστά pools στο σωστό hook.

Το ακριβές behavior εξαρτάται από Puma mode και Rails integration, άρα επιβεβαιώνεται με process-level probe.

Deadline και admission control

Το `checkout_timeout` είναι deadline αναμονής για capacity, όχι query timeout. Αν λήξει, δεν σημαίνει ότι η database είναι αργή. Μπορεί όλα τα connections να κρατιούνται από Ruby code που περιμένει HTTP, queue ή mutex. Γι' αυτό το pool timeout πρέπει να συσχετίζεται με owner stacks και transaction duration, όχι να αντιμετωπίζεται με τυφλή αύξηση του `pool`.

Η queue επιτρέπει σε waiters να πάρουν σύνδεση με σειρά αναμονής, ενώ διαθέσιμα connections αποθηκεύονται με LIFO συμπεριφορά ώστε τα πιο πρόσφατα να επαναχρησιμοποιούνται και τα παλιά να μπορούν να γίνουν idle. Αυτές οι δύο ιδιότητες δεν αντιφάσκουν. Περιγράφουν διαφορετική σειρά για demand και supply. Σε overload, το πραγματικό design question είναι πού απορρίπτεται νέα δουλειά. Ένα μικρό bounded pool με σαφές upstream admission συχνά προστατεύει καλύτερα τη database από ένα τεράστιο pool που μεταφέρει όλη την ουρά στον server.

SOURCE TRACE

Η ownership και η lifecycle logic βρίσκονται στο `connection_pool.rb`. Η wait queue βρίσκεται στο `connection_pool/queue.rb`. Ακολούθησε `lease_connection -> checkout -> acquire_connection -> Queue#poll` και την αντίστροφη διαδρομή `release_connection -> checkin -> Queue#add`.

ΠΑΓΙΔΑ

Το pool size πολλαπλασιάζεται ανά process, database configuration, role και shard που ενεργοποιείται. Μην συγκρίνεις το `pool: 10` απευθείας με database max connections χωρίς να υπολογίσεις ολόκληρη την topology.

ΜΗΧΑΝΙΣΜΟΣ

Connection capacity είναι ταυτόχρονα ownership problem και queueing problem. Η διάρκεια που ένας execution context κρατά lease έχει συχνά μεγαλύτερη σημασία από τη διάρκεια του ίδιου του SQL.

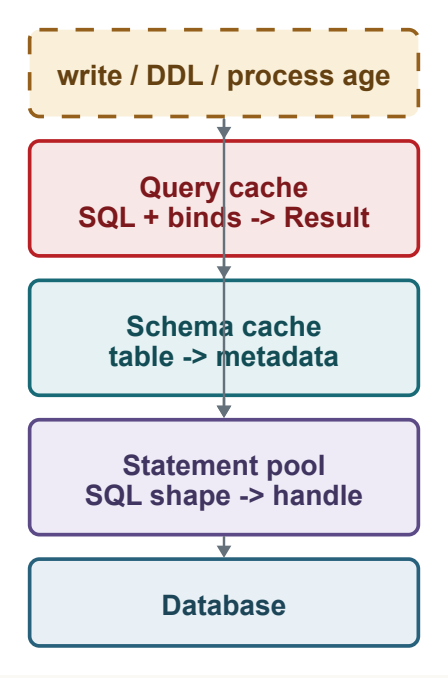
PROBE

Σε staging-like load, sample το `pool.stat` ανά role και shard κάθε δευτερόλεπτο. Κατέγραψε p95 checkout wait, query time και transaction duration ξεχωριστά. Πρόσθεσε ελεγχόμενο external sleep μέσα και έξω από `with_connection`. Η διαφορά θα δείξει αν το bottleneck είναι database ή ownership lifetime.

ΚΕΦΑΛΑΙΟ 13

Query cache, prepared statements και schema cache

Το Active Record έχει πολλά caches με παρόμοια ονόματα αλλά διαφορετικά keys, lifetimes και failure modes. Η λάθος invalidation μοιάζει συχνά με stale database.



Σχήμα 13.1 · Τα τρία cache layers γύρω από SQL execution και schema metadata.

Η query cache κρατά result sets για ίδιο SQL και binds μέσα σε περιορισμένο execution lifetime. Το prepared statement cache κρατά server-side ή adapter-side statement handles ανά SQL shape. Η schema cache κρατά columns, primary keys, indexes και data source metadata. Κανένα από αυτά δεν είναι το application fragment cache και κανένα δεν έχει το ίδιο invalidation rule.

Query cache ως execution-local read memo

Όταν είναι ενεργή, η query cache φτιάχνει key από SQL και bind attributes. Σε hit επιστρέφει duplicate result ώστε caller mutation να μην αλλάξει το cached object. Η cache registry είναι συνδεδεμένη με execution context και pools. Ο Executor ενεργοποιεί και καθαρίζει το σωστό lifecycle γύρω από request ή job.

Writes καλούν dirtying methods που καθαρίζουν query caches για relevant connections. Με πολλές databases, ένα write μπορεί να χρειαστεί να λερώσει όλες τις query caches του current thread, επειδή δεν είναι πάντα γνωστό ποιο read result εξαρτάται από το write. Μπορείς να χρησιμοποιήσεις `uncached` για block που απαιτεί fresh reads, αλλά αυτό δεν νικά replica lag ούτε το isolation snapshot μιας transaction.

```ruby
ActiveRecord::Base.uncached do
  first = Reservation.where(event_id: 42).count
  second = Reservation.where(event_id: 42).count
  puts [first, second].inspect
end
```

Το `uncached` αλλάζει μόνο αυτό το Rails layer. Αν βρίσκεσαι σε repeatable-read transaction, η database μπορεί νόμιμα να δίνει το ίδιο snapshot. Αν βρίσκεσαι σε replica, μπορεί να μην έχει εφαρμόσει ακόμη το writer WAL.

Prepared statement pool

Ο adapter μπορεί να αντιστοιχίζει SQL shape σε prepared statement name ή handle. Το pool έχει bounded size και eviction. Η PostgreSQL μπορεί να απορρίψει cached plan όταν schema change μεταβάλλει result type. Το Active Record έχει recovery paths για ορισμένες περιπτώσεις, αλλά όχι μέσα σε κάθε transaction state. Ένα rolling schema change πρέπει να σχεδιαστεί ώστε old process και new process να μην κρατούν ασύμβατα expectations.

Dynamic SQL shapes μειώνουν reuse. Διαφορετικό πλήθος literals σε raw `IN`, dynamic column lists και tenant-specific table names μπορούν να γεμίσουν το statement pool. Binds βοηθούν όταν αλλάζουν values, όχι όταν αλλάζει η ίδια η δομή SQL.

Schema cache

Η schema cache αποφεύγει repeated metadata queries κατά το boot και runtime. Μπορεί να φορτωθεί από dump και να δεθεί σε pool. Κρατά columns, columns hash, primary keys, indexes και existence information. DDL methods του adapter καθαρίζουν cache entries για tables που αλλάζουν, αλλά εξωτερικό migration ή άλλο process δεν μεταδίδει μαγικά invalidation σε ήδη ζωντανές διεργασίες.

Αυτός είναι λόγος που unsafe in-place type changes είναι επικίνδυνες σε rolling deploy. Το old process μπορεί να κρατά παλιό column metadata και prepared plan. Το new process μπορεί να γράφει νέο representation. Restart βοηθά cache state, αλλά δεν διορθώνει incompatible overlap.

```ruby
pool = ActiveRecord::Base.connection_pool
columns = pool.schema_cache.columns_hash("reservations")

puts columns.fetch("status").sql_type
```

Η introspection είναι χρήσιμη σε diagnostics. Μην κάνεις runtime application branching πάνω σε private schema cache shape. Για staged migrations, προτίμησε explicit feature state και deploy protocol.

Ένας πίνακας διάγνωσης

Σύμπτωμα	Πιθανό layer	Πρώτος έλεγχος
ίδιο SELECT, δεύτερη φορά χωρίς SQL	query cache	`CACHE` log και `uncached`
πολλά parse/prepare operations	statement pool	SQL shape cardinality
unknown ή missing column μετά DDL	schema cache	process age και cache dump
stale read μετά write	replica ή isolation	role, shard, transaction snapshot

Η διάγνωση αρχίζει με την ταυτότητα του connection. Ίδιο model name δεν σημαίνει ίδιο pool, role ή shard.

SOURCE TRACE

Η query cache υλοποίηση βρίσκεται στο `abstract/query_cache.rb`. Το bounded statement map βρίσκεται στο `statement_pool.rb` με adapter-specific subclasses. Η metadata lifecycle βρίσκεται στο `schema_cache.rb`.

ΠΑΓΙΔΑ

Το «clear the cache» είναι επικίνδυνα ασαφές. Πριν κάνεις αλλαγή, ονόμασε query, statement, schema ή application cache. Κατέγραψε key, owner, lifetime και invalidation event.

ΜΗΧΑΝΙΣΜΟΣ

Οι caches δεν είναι μία στοίβα freshness. Είναι ανεξάρτητες βελτιστοποιήσεις για results, execution plans και metadata. Ένα fresh layer μπορεί να κάθεται πάνω σε stale άλλο layer.

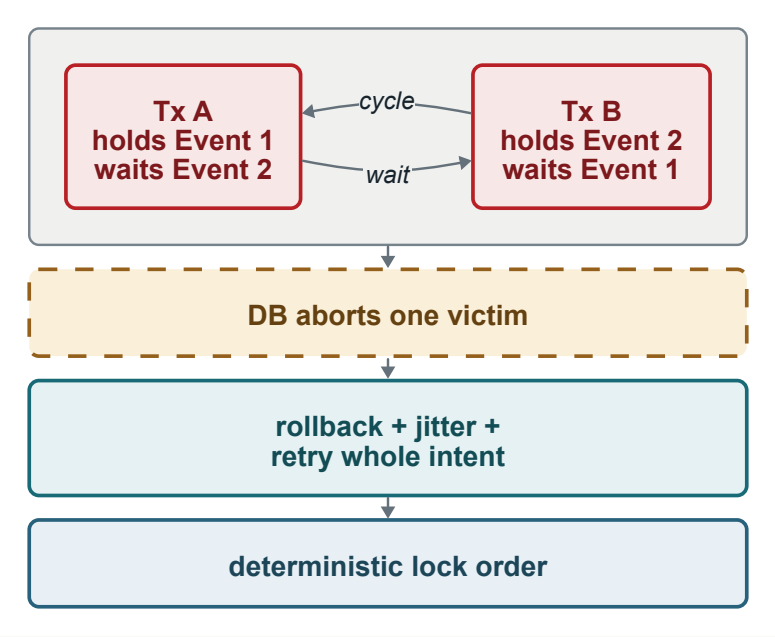
PROBE

Κάνε δύο ίδια SELECT μέσα σε request και επιβεβαίωσε query cache hit. Μετά άλλαξε bind, κάνε write και επανάλαβε. Σε ξεχωριστό disposable test database, άλλαξε additive column και σύγκρινε schema cache σε παλιό και νέο process. Μην τρέξεις destructive DDL σε production για αυτό το probe.

ΚΕΦΑΛΑΙΟ 14

Isolation, deadlocks και retry boundaries

Ένα retry είναι σωστό μόνο όταν ξανατρέχει ολόκληρη την αποτυχημένη μονάδα με ίδιο intent και χωρίς side effects που έχουν ήδη ξεφύγει.



Σχήμα 14.1 · Δύο lock orders δημιουργούν κύκλο και απαιτούν transaction-level retry.

Το Active Record ανοίγει transaction, αλλά η database ορίζει isolation, lock compatibility, deadlock detection και serialization failures. Το Rails μεταφράζει adapter errors σε exceptions όπως `ActiveRecord::Deadlocked` ή `ActiveRecord::SerializationFailure`. Η σωστή recovery unit είναι συνήθως όλη η application transaction, έξω από το transaction block.

Γιατί όχι rescue μέσα στο block

Σε PostgreSQL, statement error μπορεί να αφήσει την transaction aborted μέχρι rollback. Το να πιάσεις exception και να συνεχίσεις με άλλο query μέσα στο ίδιο block δεν αποκαθιστά valid transaction. Ακόμη και σε adapter που επιτρέπει περισσότερα, το domain state που διάβασες πριν από το conflict μπορεί να είναι stale. Το retry πρέπει να ξαναφορτώσει inputs και locks.

```ruby
class ConfirmReservation
  MAX_ATTEMPTS = 3

  def self.call(id)
    attempts = 0

    begin
      attempts += 1
      Reservation.transaction do
        reservation = Reservation.lock.find(id)
        reservation.confirm!
      end
    rescue ActiveRecord::Deadlocked, ActiveRecord::SerializationFailure
      raise if attempts >= MAX_ATTEMPTS

      sleep(0.01 * (2**attempts) + rand * 0.01)
      retry
    end
  end
end
```

Η operation πρέπει να είναι replayable. Δεν στέλνει email και δεν χρεώνει κάρτα μέσα στο retried block. Γράφει durable intent ή outbox row στην ίδια transaction. Το random jitter μειώνει synchronized retry storms. Το cap εμποδίζει infinite contention loop.

Deadlock και lock order

Deadlock υπάρχει όταν transaction A κρατά resource 1 και περιμένει 2, ενώ B κρατά 2 και περιμένει 1. Η database επιλέγει victim και το κάνει rollback. Το retry είναι safety net, όχι η βασική λύση. Η βασική λύση είναι deterministic lock order. Αν μια μεταφορά θέσεων κλειδώνει δύο events, ταξινόμησέ τα πάντα με id πριν τα κλειδώσεις.

```ruby
event_ids = [source_event_id, target_event_id].sort
events = Event.where(id: event_ids).order(:id).lock.to_a
```

Η `ORDER BY` πρέπει να αντιστοιχεί στην πραγματική σειρά acquisition του plan. Σε σύνθετα joins ή bulk updates επιβεβαίωσέ το στον adapter και στην database. Μην υποθέτεις ότι η Ruby array order αναγκάζει κάθε SQL lock order.

Isolation anomalies

Read committed, repeatable read και serializable δεν είναι levels «ποιότητας» που ανεβαίνουν δωρεάν. Αλλάζουν ποια anomalies επιτρέπονται και ποια conflicts μετατρέπονται σε abort. Το serializable συχνά απαιτεί application retry ως κανονικό control path. Pessimistic row lock προστατεύει υπάρχουσες rows, όχι αναγκαστικά την απουσία row ή arbitrary predicate. Unique constraint και exclusion constraint μπορούν να κρατήσουν invariants που ένα row lock δεν βλέπει.

Advisory locks είναι database-specific namespace locks. Το Rails core δεν τα μετατρέπει σε portable domain mutex. Αν τα χρησιμοποιήσεις με raw SQL, σχεδίασε key mapping, transaction versus session lifetime, collision domain, timeout και cleanup. Ποτέ μην κρατάς session advisory lock ενώ κάνεις unbounded network I/O.

Retry budget και observability

Ένα successful request μετά από δύο retries είναι correctness success αλλά capacity warning. Κατέγραψε exception class, attempt, operation name, lock targets ως bounded categories και συνολικό elapsed time. Μην γράφεις raw SQL με sensitive binds σε error logs. Αν το retry rate ανεβαίνει, βρες hot rows, long transactions ή inconsistent lock order.

Το timeout hierarchy πρέπει να είναι συνεπές. Database lock timeout μικρότερο από request deadline αφήνει χρόνο για controlled retry. Αν κάθε attempt μπορεί να καταναλώσει όλο το upstream timeout, το τρίτο retry είναι ήδη νεκρό work.

Lock wait και statement execution

Lock timeout και statement timeout κόβουν διαφορετικές φάσεις. Το πρώτο περιορίζει πόσο θα περιμένεις για lock. Το δεύτερο μπορεί να διακόψει και query που απέκτησε όλα τα locks αλλά κάνει ακριβό scan. Πρόσθεσε και application deadline και έχεις τρία clocks που πρέπει να αφήνουν χρόνο για cleanup και ίσως ένα retry. Αν είναι ασύνδετα, ο outer caller μπορεί να έχει ήδη εγκαταλείψει ενώ η database συνεχίζει work που κανείς δεν χρειάζεται.

Το retry budget πρέπει να είναι elapsed-time budget, όχι μόνο αριθμός attempts. Υπολόγισε το υπόλοιπο deadline πριν από backoff και πριν αρχίσεις νέα transaction. Μη χρησιμοποιείς το ίδιο policy για interactive request και offline reconciliation. Το πρώτο προστατεύει tail latency. Το δεύτερο μπορεί να περιμένει περισσότερο αλλά χρειάζεται durable progress. Και στα δύο, ένα retry που δεν χωρά να ολοκληρωθεί είναι load amplification, όχι resilience.

SOURCE TRACE

Το Rails transaction API και οι προειδοποιήσεις για rollback errors βρίσκονται στο `active_record/transactions.rb`. Η exception translation είναι adapter-specific στα `connection_adapters`. Για isolation semantics χρησιμοποίησε την τεκμηρίωση της πραγματικής database, όχι γενίκευση από άλλο adapter. Το Rails δεν εγγυάται advisory lock portability.

ΠΑΓΙΔΑ

Μην retry-άρεις `StatementInvalid` γενικά. Syntax errors, constraint violations και programmer bugs δεν είναι transient. Κράτα μικρό allowlist από συγκεκριμένα retryable exceptions και bounded attempts.

ΜΗΧΑΝΙΣΜΟΣ

Το retry boundary πρέπει να περιέχει ξανά όλες τις reads και writes που σχηματίζουν την απόφαση, αλλά κανένα μη αναστρέψιμο external effect. Αυτό είναι application contract, όχι adapter option.

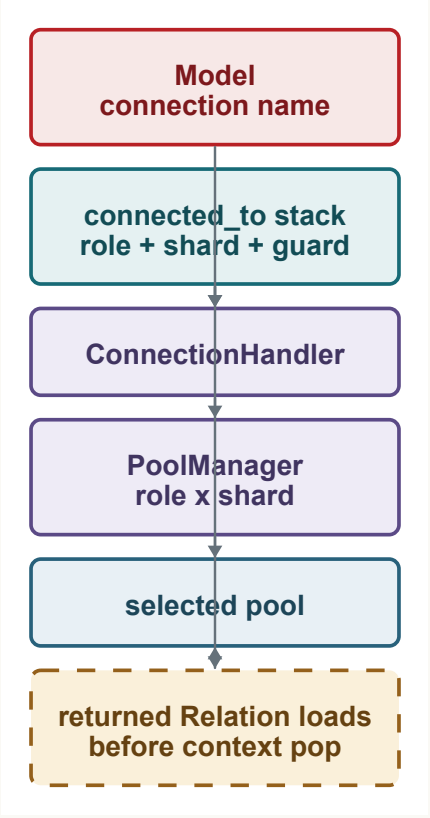
PROBE

Σε test database, άνοιξε δύο connections και κλείδωσε τα ίδια δύο rows σε αντίθετη σειρά με barriers ώστε να δημιουργήσεις deterministic deadlock. Έλεγξε ότι μία transaction αποτυγχάνει, το outer retry ξαναφορτώνει state και το τελικό invariant ισχύει. Κατέγραψε attempts και connection ids.

ΚΕΦΑΛΑΙΟ 15

Roles, shards και execution-local context

Το model δεν «συνδέεται σε μια βάση». Επιλύει connection name, current role και current shard μέσα στο execution context και βρίσκει το αντίστοιχο pool.



Σχήμα 15.1 · Η επιλογή connection περνά από context stack και PoolManager.

Το `ConnectionHandler` κρατά map από connection name σε `PoolManager`. Κάθε PoolManager κρατά configurations με δύο dimensions: role και shard. Το model δίνει το connection specification name του. Το current execution δίνει role και shard. Ο handler επιστρέφει το pool στη διασταύρωση. Έτσι δύο calls του ίδιου model μπορούν να χρησιμοποιήσουν διαφορετικό socket set χωρίς να αλλάξει η model class.

Η connected_to stack

Το `connected_to` προσθέτει entry σε stack μέσα στο `ActiveSupport::IsolatedExecutionState`. Το entry έχει role, shard, `prevent_writes` και classes στις οποίες εφαρμόζεται. Σε ensure το entry βγαίνει. Τα `current_role` και `current_shard` διαβάζουν τη stack από το τέλος και βρίσκουν το πρώτο relevant entry. Nested switching είναι λοιπόν dynamically scoped context, όχι global αλλαγή configuration.

```ruby
AnimalsRecord.connected_to(role: :reading, shard: :europe) do
  puts AnimalsRecord.current_role
  puts AnimalsRecord.current_shard
  Animal.limit(10).load
end
```

Η block μορφή είναι κρίσιμη επειδή δίνει cleanup. Η `connecting_to` χωρίς block προσθέτει state που προορίζεται για console-like use και δεν συνιστάται μέσα σε request. Exception χωρίς ensure θα μπορούσε διαφορετικά να αφήσει τον επόμενο κώδικα σε λάθος tenant ή writer.

Το lazy relation trap που το Rails κλείνει

Υπάρχει μια βαθιά λεπτομέρεια στο `with_role_and_shard`: αν το block επιστρέψει `ActiveRecord::Relation`, το Rails καλεί `load` πριν αφαιρέσει το context. Χωρίς αυτό, το lazy relation θα εκτελούνταν αργότερα πάνω στο προηγούμενο role ή shard. Το returned object παραμένει relation, αλλά έχει materialized records.

Αυτό δεν σώζει κάθε lazy object. Enumerator, association proxy, proc ή custom query wrapper μπορεί να ξεφύγει από το block και να κάνει I/O μετά το pop. Το boundary contract παραμένει: materialize ό,τι ανήκει στο switched context πριν βγεις.

```ruby
relation = ActiveRecord::Base.connected_to(role: :reading) do
  Reservation.where(event_id: 42)
end

puts relation.loaded?
```

Prevent writes είναι guardrail

Το reading role ενεργοποιεί `prevent_writes`. Ο adapter εξετάζει SQL και απορρίπτει queries που ταξινομεί ως writes. Αυτό προστατεύει από accidental Active Record writes. Δεν ισοδυναμεί με database user που έχει μόνο SELECT, δεν είναι security boundary και δεν εγγυάται ότι κάθε vendor-specific write θα αναγνωριστεί. Η πραγματική replica πρέπει να έχει database permissions και topology που επιβάλλουν τον ρόλο της.

Read-your-writes και automatic switching

Το database selector middleware μπορεί να στείλει recent writers στον primary για configurable delay και άλλες reads στη replica. Αυτό είναι heuristic γύρω από πιθανό replication lag. Δεν μετρά

απαραίτητα πραγματικό replay position. Για operation που πρέπει να δει αμέσως δικό της write, μείνε στον writer ή χρησιμοποίησε explicit consistency token που υποστηρίζει η υποδομή σου.

Με shards, το tenant routing είναι correctness boundary. Χρησιμοποίησε `prohibit_shard_swapping` αφού επιλεγεί shard για request, ώστε nested code να μην αλλάξει σιωπηρά tenant. Όμως το guard δεν επιβεβαιώνει ότι η πρώτη επιλογή ήταν σωστή. Το authentication-to-shard mapping χρειάζεται δικό του test.

Capacity πολλαπλασιάζεται

Κάθε ενεργό `(connection name, role, shard)` έχει δικό του pool. Lazy pool activation περιορίζει one-off κόστος, αλλά worker που αγγίζει πολλούς shards μπορεί να ανοίξει πολλά pools. Μέτρα πραγματικά activated pools και database connections ανά process. Το theoretical product όλων των dimensions είναι worst case, όχι πάντα measured usage.

Context stack και deferred work

Η connected stack είναι scoped στο current isolated execution state. Αυτό προστατεύει concurrent requests, αλλά δεν μεταφέρει αυτόματα την πρόθεσή σου σε νέο thread, future ή job. Αν scheduled work πρέπει να τρέξει σε συγκεκριμένο shard, serialize το tenant ή shard identity ως explicit input και ξανακάνε το routing μέσα στο execution. Μην αντιγράψεις ολόκληρο process-local stack σε durable payload.

Πρόσεξε επίσης class scope. Μια abstract record class μπορεί να έχει διαφορετικό connection handler contract από άλλη hierarchy. Το ότι δύο models βρίσκονται στο ίδιο block δεν αποδεικνύει ότι χρησιμοποιούν το ίδιο pool ή shard mapping. Σε cross-database workflow κατέγραψε για κάθε statement connection name, role, shard και database transaction id όπου υποστηρίζεται. Έτσι φαίνεται αν η εφαρμογή έκανε πραγματικά μία atomic operation ή δύο ανεξάρτητα commits που χρειάζονται compensation.

Session state ακολουθεί το connection

Temporary tables, advisory locks, transaction isolation changes και άλλα session-level features ανήκουν στο συγκεκριμένο database connection. Το `connected_to` επιλέγει pool. Δεν εγγυάται ότι δύο ξεχωριστά blocks θα πάρουν το ίδιο socket, ούτε ότι ένα session setting θα ακολουθήσει logical request. Αν χρειάζεσαι τέτοιο state, κράτησε explicit `with_connection` scope, κάνε cleanup σε `ensure` και προτίμησε transaction-local database settings όπου υπάρχουν.

Αυτό γίνεται κρίσιμο με replica promotion. Το application role `:reading` δεν σημαίνει από μόνο του read-only physical server για πάντα. Topology manager ή DNS μπορεί να αλλάξει endpoint. Αν correctness εξαρτάται από πραγματικό WAL position, χρησιμοποίησε mechanism της database και όχι το συμβολικό Rails role ως proof.

Σε shard migration, παλιός και νέος worker μπορεί να διαφωνούν για το tenant mapping. Κάνε το mapping versioned και observable. Ένα ασφαλές move συνήθως χρειάζεται phase όπου writes έχουν μοναδικό owner, reads μπορούν να συγκριθούν και jobs μεταφέρουν tenant identity που επιλύεται με συμβατό τρόπο. Το connection context βρίσκει πού θα τρέξει query. Δεν αποφασίζει μόνο του πότε η μεταφορά ownership ολοκληρώθηκε.

SOURCE TRACE

Η επιλογή pool βρίσκεται στα `connection_handler.rb` και `pool_manager.rb`. Η execution-local stack και το eager load του returned Relation βρίσκονται στα `core.rb` και `connection_handling.rb`. Το public setup περιγράφεται στο Multiple Databases guide.

ΠΑΓΙΔΑ

Μην περνάς lazy enumerator ή proc έξω από `connected_to` block αν μπορεί να κάνει query. Το Rails materializes returned Relation ειδικά, όχι κάθε πιθανό lazy abstraction της εφαρμογής.

ΜΗΧΑΝΙΣΜΟΣ

Role και shard είναι μέρος του execution context. Κάθε query trace που τα αγνοεί είναι ελλιπές, ακόμη αν το SQL text είναι σωστό.

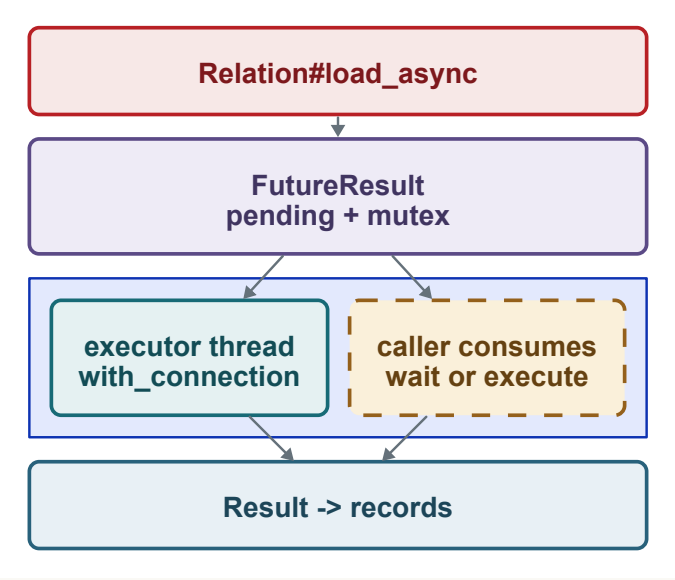
PROBE

Κάνε subscribe στο `sql.active_record` και πρόσθεσε sanitized role, shard και connection object id. Επέστρεψε Relation, Enumerator και proc από switched block σε test. Επιβεβαίωσε πότε γίνεται το query. Μετά έλεγξε ότι nested shard switch απορρίπτεται μέσα σε `prohibit_shard_swapping`.

ΚΕΦΑΛΑΙΟ 16

Async queries και το πραγματικό resource budget

Το `load_async` επικαλύπτει αναμονή μόνο όταν υπάρχει ελεύθερος executor και δεύτερο database connection. Δεν κάνει τη database γρηγορότερη.



Σχήμα 16.1 · Το FutureResult μπορεί να τρέξει background ή να ολοκληρωθεί foreground.

Όταν καλείς `relation.load_async`, το relation αποκτά connection προσωρινά για να ελέγξει αν async execution είναι enabled. Αν δεν είναι, κάνει κανονικό load. Αν είναι και το relation δεν έχει loaded, καλεί το main query path με async flag μόνο όταν δεν βρίσκεται σε joinable transaction. Το result μπορεί να είναι array, αν εκτελέστηκε αμέσως, ή `FutureResult`, αν προγραμματίστηκε.

Το relation σημειώνεται loaded ακόμη όταν κρατά future. Όταν ζητήσεις records, το `exec_queries` παίρνει το result. Αν η background δουλειά έχει τελειώσει, επιστρέφει αμέσως. Αν όχι, ο foreground caller περιμένει. Αν ο executor δεν πρόλαβε να αρχίσει, ο foreground path μπορεί να αποκτήσει το mutex και να εκτελέσει ο ίδιος το query. Έτσι η correctness δεν εξαρτάται από το να ξεκινήσει οπωσδήποτε background thread.

FutureResult και Promise

Το `FutureResult` κρατά pool, query arguments, mutex, pending state, result ή error. Το background path κάνει `pool.with_connection`, αποκτά mutex με `try_lock` και εκτελεί. Το foreground `result`

περιμένει στο ίδιο mutex ή εκτελεί αν παραμένει pending. Τα instrumentation events buffer-άρονται και δημοσιεύονται στο consuming context με lock wait metadata.

Το `ActiveRecord::Promise` είναι lazy transformation πάνω στο future. Το `then` συνθέτει block, αλλά το block δεν είναι γενικός async callback scheduler. Τρέχει όταν ζητηθεί value. Αν περιμένεις promise και αμέσως καλείς `.value`, δεν έχεις δώσει χρήσιμη δουλειά να επικαλυφθεί.

```ruby
reservations = Reservation.confirmed.where(event_id: 42).load_async
payments = Payment.where(event_id: 42).async_sum(:amount_cents)

summary = EventSummary.compute_without_database(event_id: 42)

puts reservations.to_a.length
puts payments.value
puts summary
```

Η middle δουλειά πρέπει να είναι ανεξάρτητη. Αν χρειάζεται result του πρώτου query, η αλληλουχία είναι αναγκαστικά sequential.

Transactions και consistency

Async query δεν μπορεί να χρησιμοποιήσει άλλο connection και ταυτόχρονα να δει το uncommitted state της current transaction. Γι' αυτό το relation path δεν προγραμματίζει background query μέσα σε joinable transaction. Ακόμη αν χειροποίητος κώδικας βρει τρόπο να χρησιμοποιήσει δεύτερο connection, θα αλλάξει το isolation boundary. Αυτό δεν είναι optimization. Είναι διαφορετικά semantics.

Ο λογαριασμός των connections

Με `global_thread_pool`, πολλά pools μοιράζονται query executor threads αλλά κάθε query χρειάζεται connection από το δικό του pool. Με `multi_thread_pool`, κάθε pool μπορεί να έχει δικό του executor. Τα config names και defaults πρέπει να επαληθεύονται στην έκδοση της εφαρμογής. Το capacity equation είναι περίπου:

```output
foreground DB concurrency
+ background async queries actually in flight
+ job and Cable DB concurrency
<= available connections across the relevant pools
```

Αν web threads γεμίζουν pool και καθένα προγραμματίσει async query, οι background workers περιμένουν connections. Οι foreground callers φτάνουν `.to_a` και ίσως περιμένουν τα futures. Έχεις μετακινήσει την queue, όχι μειώσει το work.

Πότε αξίζει

Αξίζει όταν δύο ή περισσότερα independent, latency-heavy queries μπορούν να τρέξουν παράλληλα, η database έχει spare capacity και υπάρχει χρήσιμη CPU ή I/O δουλειά ανάμεσα στο schedule και

consume. Δεν αξίζει για μία γρήγορη query, για hot database στο saturation ή για queries που ανταγωνίζονται το ίδιο lock.

Μέτρα wall time, total database time, connection wait και database load. Η wall time μπορεί να πέσει ενώ total DB work μένει ίδιο ή αυξάνει. Αυτό είναι αποδεκτό μόνο αν το συνολικό σύστημα έχει capacity.

Completion thread και context

Το object που επιστρέφεται μοιάζει με promise, αλλά μην προβάλλεις πάνω του γενικό JavaScript promise model. Η query έχει συγκεκριμένο Active Record executor, connection-pool lifecycle και foreground fallback. Αν ο caller ζητήσει το αποτέλεσμα πριν αρχίσει το background work, μπορεί να κερδίσει τη race και να εκτελέσει ο ίδιος. Άρα thread names στα traces δεν είναι σταθερό public contract.

Οι transformations πάνω σε Active Record promise διατηρούν lazy composition, όμως application code που χρειάζεται request-local metadata πρέπει να ελέγξει πού εκτελείται. Μη βασίζεις authorization ή tenant selection σε implicit thread local που ίσως λείπει. Κλείσε αυτά τα decisions μέσα στο Relation πριν από το schedule και πέρασε observability context με υποστηριζόμενο execution mechanism. Το αποτέλεσμα μπορεί να είναι deferred. Η ασφάλεια του query shape δεν πρέπει να είναι deferred μαζί του.

SOURCE TRACE

Η public είσοδος βρίσκεται στο `Relation#load_async`. Η race ανάμεσα σε background και foreground execution βρίσκεται στο `future_result.rb`, ενώ το lazy transformation API είναι στο `promise.rb`. Ο executor δημιουργείται από το connection pool.

ΠΑΓΙΔΑ

Μην αυξήσεις async executor threads χωρίς να επανυπολογίσεις database connection budget. Thread χωρίς connection δεν εκτελεί query. Περισσότερα waiters μπορούν να κάνουν tail latency χειρότερο.

ΜΗΧΑΝΙΣΜΟΣ

Async query είναι latency overlap mechanism με πρόσθετο connection demand. Το σωστό success metric είναι request wall time μέσα σε αποδεκτό database load, όχι η παρουσία `ASYNC` στο log.

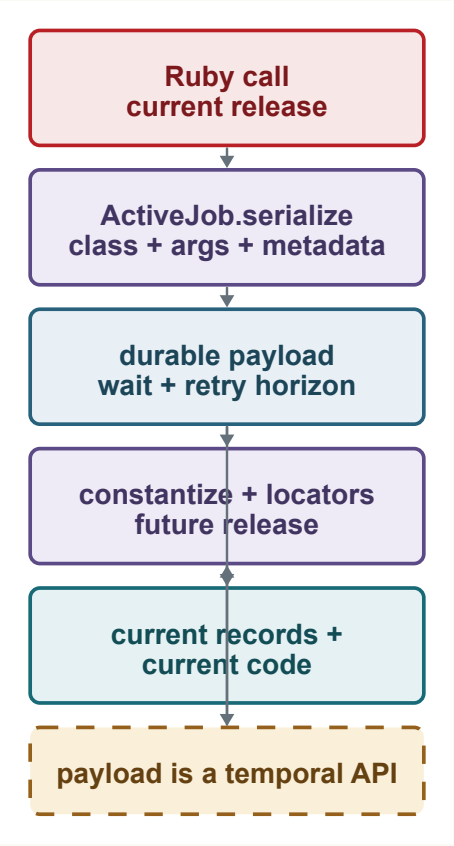
PROBE

Σε representative dataset, σύγκρινε sequential και async εκτέλεση δύο independent queries. Κατέγραψε wall time, άθροισμα SQL times, `pool.stat[:waiting]` και database CPU. Επανάλαβε με pool κοντά στο saturation. Κράτα το async version μόνο αν βελτιώνει tail latency χωρίς να μεταφέρει υπερβολικό κόστος στους άλλους requests.

ΚΕΦΑΛΑΙΟ 17

Active Job serialization και GlobalID

Ένα enqueued job είναι versioned μήνυμα που θα διαβαστεί αργότερα από πιθανώς διαφορετικό release. Δεν είναι frozen Ruby method call.



Σχήμα 17.1 · Το Ruby call γίνεται payload και ξαναχτίζεται σε μελλοντικό process.

Το `perform_later` δημιουργεί job instance με `job_id`, queue, priority, arguments, locale, timezone και counters. Πριν περάσει στον adapter, το `serialize` παράγει hash. Το hash περιέχει το string name της job class και serialized arguments. Όταν worker το πάρει, το `ActiveJob::Base.deserialize` κάνει safe constantize του σημερινού class name, δημιουργεί σημερινό instance και επαναφέρει state. Άρα producer code και consumer code δεν χρειάζεται να είναι το ίδιο release.

Αυτό είναι το θεμέλιο του job compatibility problem. Rename μιας job class, αλλαγή arity, αφαίρεση keyword ή αλλαγή custom serializer μπορεί να σπάσει ήδη αποθηκευμένα payloads. Το queue depth μετατρέπει ένα deploy σε temporal API.

Τι ακριβώς ταξιδεύει

Intrinsic values όπως numbers, strings, arrays και hashes serializes recursively. Symbols, dates, durations και άλλα supported types χρησιμοποιούν registered serializers. Hash keys πρέπει να είναι strings ή symbols και ορισμένα `_aj_` keys είναι reserved. Άγνωστος object type χρειάζεται custom serializer ή GlobalID identification.

```ruby
class ConfirmReservationJob < ApplicationJob
  def perform(reservation_id, requested_by_id:, contract_version: 1)
    reservation = Reservation.find(reservation_id)
    actor = User.find(requested_by_id)
    Reservations::Confirm.call(
      reservation: reservation,
      actor: actor,
      contract_version: contract_version
    )
  end
end
```

Τα explicit IDs κάνουν το message schema ορατό. Το default argument επιτρέπει σε παλιό payload να διαβαστεί από νέο code. Η version δεν χρειάζεται να αυξάνεται σε κάθε refactor. Χρειάζεται όταν αλλάζει η σημασία του payload.

GlobalID είναι reference, όχι snapshot

Αν περάσεις `reservation` αντί για id, το Active Job αποθηκεύει GlobalID URI με app, model name και model id. Κατά την εκτέλεση, ο locator φορτώνει record από τη σημερινή database. Δεν αποθηκεύει attributes της στιγμής enqueue. Αν το status άλλαξε, το job βλέπει νέο status. Αν το row διαγράφηκε, deserialization μπορεί να αποτύχει πριν μπει στη `perform`.

Αυτό είναι συχνά το σωστό contract για «κάνε κάτι με το current record». Δεν είναι σωστό για «στείλε ακριβώς την τιμή που ίσχυε κατά την αγορά». Εκεί χρειάζεται immutable snapshot fields ή domain event record. Η επιλογή ID versus snapshot είναι business semantics, όχι serialization preference.

Signed GlobalID προσθέτει authenticity, purpose και προαιρετική λήξη. Δεν κάνει το record immutable και δεν εγγυάται ότι υπάρχει. Είναι χρήσιμο όταν το reference περνά από μη έμπιστο caller ή έχει χρονικό window.

Custom serializers ως deployed protocol

Ένας custom serializer γράφει discriminator και fields και αργότερα ξαναχτίζει object. Το class name του serializer αποθηκεύεται στο payload. Αν αφαιρεθεί, παλιά jobs δεν διαβάζονται. Κράτα serializer backward-readable για τουλάχιστον το μέγιστο queue age συν retry horizon. Για μεγάλες αλλαγές, πρόσθεσε new serializer version αντί να αλλάξεις σιωπηρά την παλιά representation.

Η deserialization τυλίγει failures σε `ActiveJob::DeserializationError`. Το `discard_on` είναι σωστό μόνο όταν το job έχει γίνει οριστικά irrelevant, όπως deleted subject. Αν failure προέρχεται από προσωρινή database διαθεσιμότητα, discard χάνει δουλειά. Η έκδοση GlobalID 1.4 ξεχωρίζει σε δικό της locator API record not found από record unavailable, αλλά το Active Job contract και ο adapter path πρέπει να ελεγχθούν πριν στηριχτεί policy πάνω στη λεπτομέρεια.

Payload inventory πριν από αλλαγή

Ένα safe deploy ξέρει oldest enqueued job class, μέγιστο retry age και scheduled horizon. Recurring task μπορεί να παράγει νέο payload αμέσως μετά το deploy, ενώ delayed job μπορεί να ξυπνήσει μήνες μετά. Αν δεν μπορείς να επιθεωρήσεις payloads λόγω adapter abstraction, κράτα explicit compatibility window ως operational contract.

SOURCE TRACE

Το envelope δημιουργείται στο `active_job/core.rb` και τα arguments στο `active_job/arguments.rb`. Η reference resolution βρίσκεται στο tagged `globalid`. Ακολούθησε `serialize -> Arguments.serialize -> GlobalID -> adapter -> deserialize -> Locator`.

ΠΑΓΙΔΑ

Μην περνάς mutable model επειδή φαίνεται πιο object-oriented. Το payload κρατά μόνο reference. Γράψε σε μία πρόταση αν το job χρειάζεται current state ή historical snapshot και διάλεξε representation από αυτό.

ΜΗΧΑΝΙΣΜΟΣ

Το job payload είναι protocol ανάμεσα σε χρόνους και releases. Class names, argument shapes, serializers και domain meaning χρειάζονται backward compatibility όσο υπάρχει έστω ένα παλιό message.

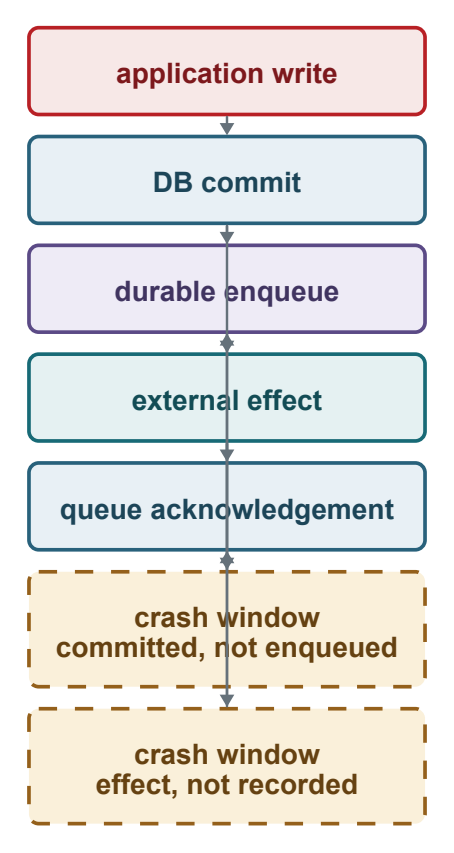
PROBE

Πάρε πραγματικό serialized hash από κάθε critical job χωρίς sensitive values. Αποθήκευσέ το ως fixture και κάνε deserialize με το νέο release. Δοκίμασε deleted GlobalID subject, renamed class compatibility και παλιό contract version. Το test πρέπει να καλεί `perform_now` πάνω στο deserialized instance.

ΚΕΦΑΛΑΙΟ 18

Enqueue after commit, retries και idempotency

To enqueue timing, το retry policy και το idempotency ledger είναι τρία διαφορετικά mechanisms. Κανένα μόνο του δεν δίνει exactly-once effect.



Σχήμα 18.1 · Τα crash windows ανάμεσα σε commit, enqueue, perform και acknowledgement.

Το Rails μπορεί να αναβάλει enqueue μέχρι να κάνουν commit όλες οι ανοιχτές Active Record transactions. Το `EnqueueAfterTransactionCommit` τυλίγει το raw enqueue και καταχωρίζει block μέσω `ActiveRecord.after_all_transactions_commit`. Σε rollback, το enqueue δεν πρέπει να γίνει. Σε commit, ο adapter καλείται μετά.

Αυτό λύνει το κλασικό race όπου worker βρίσκει job πριν γίνει visible το record. Δεν κάνει enqueue και application write atomic όταν η queue βρίσκεται σε άλλη database ή υπηρεσία. Υπάρχει window: η

application database έκανε commit και η process πέθανε πριν ο adapter αποθηκεύσει το job. Για απαραίτητη δουλειά, το outbox παραμένει ο durable bridge.

```ruby
class IssueTicketJob < ApplicationJob
  self.enqueue_after_transaction_commit = true

  retry_on Net::OpenTimeout,
    wait: :polynomially_longer,
    attempts: 8,
    jitter: 0.20

  def perform(reservation_id)
    TicketIssuance.call(reservation_id: reservation_id)
  end
end
```

Τι κάνει πραγματικά το retry_on

Το `retry_on` δηλώνει `rescue_from` handler. Οι handlers αναζητούνται από κάτω προς τα πάνω στη declaration και inheritance chain. Για κάθε declaration το job κρατά ξεχωριστό execution counter στο serialized payload. Αν υπάρχουν attempts διαθέσιμα, το handler υπολογίζει delay, προαιρετικά αναφέρει το error και καλεί `retry_job`. Όταν εξαντληθούν, τρέχει block αν δόθηκε ή ξανασηκώνει.

Το `:polynomially_longer` στην έκδοση αναφοράς χρησιμοποιεί τέταρτη δύναμη του execution count, jitter και σταθερό offset. Μη γράφεις runbook με πρόχειρο «exponential» label. Υπολόγισε πραγματικό retry horizon για τις ρυθμίσεις σου. Ο queue backend μπορεί να έχει επιπλέον retry ή redelivery behavior έξω από το Active Job. Τα δύο layers δεν μοιράζονται αναγκαστικά counter.

Idempotency key και effect ledger

At-least-once delivery σημαίνει ότι ίδιο intent μπορεί να φτάσει ξανά: timeout μετά το effect αλλά πριν το acknowledgement, worker crash, manual retry ή queue recovery. Το job πρέπει να αναγνωρίζει stable business identity. Το `job_id` δεν αρκεί αν application code δημιουργήσει νέο job για ίδιο intent.

```ruby
class TicketIssuance
  def self.call(reservation_id:)
    issuance = TicketIssue.create_or_find_by!(
      reservation_id: reservation_id,
      kind: "final_ticket"
    )

    return if issuance.completed?

    Tickets::Deliver.call(issuance: issuance)
  end
end
```

Χρειάζεται unique database constraint στα `(reservation_id, kind)`. Ακόμη μένει external ambiguity: αν το provider παρέδωσε και το connection χάθηκε πριν την απάντηση, το local `completed` είναι false. Ιδανικά το provider δέχεται δικό μας idempotency key. Διαφορετικά χρειάζεται query/reconciliation protocol, όχι blind repeat.

Retry taxonomy

Retryable είναι transient unavailability, rate limit με σαφές delay, deadlock γύρω από replayable transaction ή timeout με safe idempotency. Non-retryable είναι invalid payload, παραβίαση domain contract, missing permanently deleted subject και programmer error. Unknown failure δεν γίνεται αυτόματα retryable επειδή το queue μπορεί να ξανατρέξει.

`discard_on` πρέπει να παράγει observable outcome. Ένα silently discarded job μπορεί να είναι σωστό, αλλά operations χρειάζεται count και reason. Το `after_discard` είναι σημείο για report ή metric, όχι για να επιχειρήσει ξανά το ίδιο unsafe effect.

Shutdown και redelivery

Graceful shutdown δίνει χρόνο στα active jobs. Αν λήξει το timeout, process μπορεί να πεθάνει στη μέση. Το queue backend πρέπει να αναγνωρίσει orphaned claim ή να επαναφέρει job. Το application δεν πρέπει να υποθέτει ότι method return και acknowledgement είναι atomic. Idempotency καλύπτει ακριβώς αυτό το τελευταίο κενό.

Handler order και protocol evolution

Τα retry και discard handlers αναζητούνται από το πιο πρόσφατα δηλωμένο προς τα παλιότερα και μέσα στην inheritance chain. Ένα broad `retry_on StandardError` μπορεί έτσι να σκιάσει πιο συγκεκριμένη πολιτική που νόμιζες ότι εφαρμόζεται. Κράτα τα handlers μικρά, ordered από intention και βάλε characterization test που ελέγχει wait, attempts και τελικό outcome για κάθε exception class.

Το job id δεν είναι πάντα το σωστό idempotency key. Κάθε retry του ίδιου Active Job κρατά identity, αλλά duplicate business command μπορεί να έρθει με νέο job id από HTTP retry, reconciler ή manual replay. Το key πρέπει να ονομάζει το effect, όπως `capture/payment_42/v1`. Βάλε version όταν αλλάζει η σημασία του effect. Διαφορετικά ένα παλιό ledger row μπορεί να κάνει skip μια νέα, νόμιμα διαφορετική operation μετά από deploy.

Enqueue failure και perform failure

Η αποτυχία πριν το backend δεχτεί payload είναι διαφορετική από exception μέσα στο `perform`. Η πρώτη συμβαίνει στο producer path και μπορεί να αφήσει committed domain intent χωρίς job. Η δεύτερη είναι ήδη queue-owned execution και περνά από retry policy. Μην τις ενώσεις σε ένα metric `job_failed`. Χρειάζονται διαφορετικό owner, alert και recovery query.

Ακόμη και successful adapter return δεν είναι end-to-end απόδειξη. Σημαίνει ό,τι υπόσχεται ο συγκεκριμένος adapter για durable acceptance. Κατέγραψε Active Job id και provider job id όταν υπάρχει, αλλά κράτα το domain intent id ως σταθερό νήμα σε manual replay ή migration σε νέο backend.

Backoff ως distributed load control

Το exponential backoff με jitter δεν είναι μόνο ευγένεια προς dependency. Αποσυγχρονίζει jobs που απέτυχαν μαζί, ώστε η ανάκαμψη να μη δημιουργήσει νέο stampede. Το maximum wait πρέπει να χωρά στο business deadline και το total attempt horizon να φαίνεται σε operations. Για rate limit με server-provided retry time, χρησιμοποίησε bounded adapter που ελέγχει και κανονικοποιεί την τιμή αντί να εμπιστεύεται αυθαίρετο header.

Σε deploy, παλιό retry payload μπορεί να ξυπνήσει μετά από εβδομάδες. Το error handler, ο serializer και το idempotency ledger πρέπει να παραμένουν συμβατά για ολόκληρο αυτό το horizon. Αυτό είναι release constraint, όχι μόνο queue setting.

SOURCE TRACE

Η deferral logic βρίσκεται στο `enqueue_after_transaction_commit.rb`. Το `retry_on`, τα counters, το jitter και το handler order βρίσκονται στο `active_job/exceptions.rb`. Το public behavior περιγράφεται στο `Active Job Basics`.

ΠΑΓΙΔΑ

`enqueue_after_transaction_commit` δεν είναι transactional outbox. Κλείνει το visibility race πριν από commit, αλλά αφήνει crash window μετά το commit και πριν από durable enqueue όταν τα δύο stores δεν μοιράζονται transaction.

ΜΗΧΑΝΙΣΜΟΣ

Retry αποφασίζει πότε ξαναδοκιμάζουμε. Idempotency αποφασίζει αν η επανάληψη αλλάζει το αποτέλεσμα. Reconciliation αποφασίζει τι κάνουμε όταν δεν ξέρουμε αν το εξωτερικό effect συνέβη.

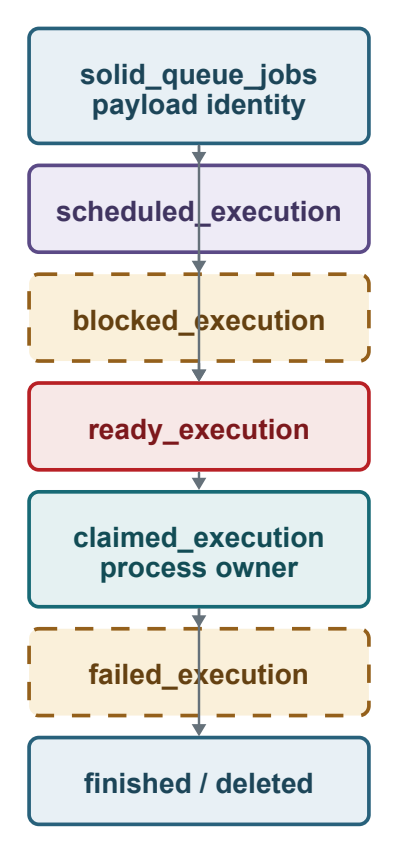
PROBE

Σε integration test βάλε barriers στα τέσσερα σημεία: πριν commit, μετά commit πριν enqueue, μετά external effect πριν local completion και μετά perform πριν acknowledgement. Προσομοίωσε crash σε κάθε σημείο. Μέτρησε τελικά business effects, όχι πόσες φορές μπήκε η `perform`.

ΚΕΦΑΛΑΙΟ 19

Το table state machine του Solid Queue

Στο Solid Queue η κατάσταση ενός job δεν είναι enum σε μία row. Εκφράζεται από job row και από την παρουσία του σε διαφορετικούς execution tables.



Σχήμα 19.1 · Οι μετακινήσεις scheduled, blocked, ready, claimed, failed και finished.

Το `solid_queue_jobs` κρατά το serialized Active Job payload και βασικά metadata. Ξεχωριστοί πίνακες αναπαριστούν eligibility και execution state: `scheduled_executions`, `ready_executions`, `blocked_executions`, `claimed_executions` και `failed_executions`. Υπάρχουν ακόμη semaphores, processes, recurring tasks, recurring executions και pauses. Τα indexes είναι μέρος του scheduler design, όχι απλή database hygiene.

Μετά τη δημιουργία job, το `prepare_for_execution` κοιτά αν είναι due. Future job αποκτά scheduled execution. Due job χωρίς concurrency limit αποκτά ready execution. Due job με limit προσπαθεί να πάρει semaphore. Αν δεν μπορεί, αποκτά blocked execution. Η job row παραμένει η ταυτότητα ενώ οι execution rows αλλάζουν.

Scheduled προς ready

Ο dispatcher κάνει polling και καλεί `ScheduledExecution.dispatch_next_batch`. Επιλέγει due rows σε bounded batch, τις προωθεί σε ready ή blocked ανά concurrency rule και αφαιρεί τις scheduled representations. Αυτό σημαίνει ότι η ακρίβεια ενός scheduled time εξαρτάται από polling interval, dispatcher availability, batch backlog και database contention. Είναι lower bound eligibility, όχι hard real-time deadline.

Ready προς claimed

Ο worker ζητά μέχρι τόσα executions όσα idle threads έχει. Το `ReadyExecution.claim` χτίζει relations ανά selected queue, επιλέγει ordered candidates με non-blocking lock, δημιουργεί claimed executions δεμένα στο process και διαγράφει τις αντίστοιχες ready rows. Η transaction γύρω από αυτή τη μετακίνηση αποτρέπει δύο healthy workers να claim-άρουν την ίδια ready row.

OUTPUT

```
job row:         exists throughout active lifecycle
ready row:       eligible, not owned
claimed row:     owned by registered process
failed row:      terminal failure awaiting action
finished_at:     completed if finished jobs are preserved
```

Το non-blocking lock αφήνει έναν worker να παρακάμψει rows που άλλος worker κλειδώνει. Αυτό βελτιώνει throughput αλλά αφαιρεί strict global ordering. Queue order, numeric priority και creation time επηρεάζουν candidate ordering. Δεν αποτελούν exactly-in-order delivery contract υπό concurrency.

Perform και outcome

Το claimed execution εκτελεί το Active Job μέσω worker thread pool. Σε success, το job γίνεται finished ή διαγράφεται ανά configuration και το concurrency lock απελευθερώνεται. Σε application failure, δημιουργείται failed execution όταν δεν έχει ήδη γίνει retry από Active Job. Manual retry ξαναπροετοιμάζει το job για execution.

Αν process εξαφανιστεί, supervisor maintenance εντοπίζει stale process row και τα orphaned claims αποτυγχάνουν με process-missing error. Αυτό κάνει το failure visible. Δεν μπορεί να γνωρίζει αν η business operation ολοκλήρωσε λίγο πριν τον θάνατο. Το application-level idempotency παραμένει απαραίτητο.

Queue database ως production dependency

Η database-backed queue έχει γνωστά transactional και operational εργαλεία, αλλά δεν είναι «δωρεάν». Polling, claim locks, cleanup, indexes και finished-job retention δημιουργούν write load. Αν χρησιμοποιεί ξεχωριστή database, έχει ξεχωριστό pool και failure domain. Αν μοιράζεται application database, queue contention μπορεί να επηρεάσει requests και κάποιες atomicity επιλογές αλλάζουν.

Μην αλλάζεις schema των internal tables χειροκίνητα από intuition. Χρησιμοποίησε τις migrations της έκδοσης Solid Queue και έλεγξε query plans πάνω στο δικό σου backlog distribution. Queue wildcard selection μπορεί να έχει διαφορετικό plan κόστος από explicit queues, ειδικά σε databases όπου prefix matching απαιτεί διαφορετικό query.

Τα σωστά metrics είναι state transitions

Το συνολικό job count δεν αρκεί. Μέτρα ready age, scheduled lateness, blocked age ανά bounded job class ή concurrency group, claim-to-finish duration, failed rate και stale process heartbeats. Ένα queue με λίγα jobs μπορεί να έχει ένα blocked critical job για ώρες. Ένα queue με πολλά short jobs μπορεί να είναι υγιές.

SOURCE TRACE

Η έκδοση αναφοράς είναι το tag `solid_queue v1.5.0`. Δες το generated `queue_schema.rb`, μετά `Job::Executable`, `Job::Schedulable`, `ScheduledExecution`, `ReadyExecution` και `ClaimedExecution`. Το κρίσιμο claim trace είναι `Worker#claim_executions -> ReadyExecution.claim -> select_candidates -> ClaimedExecution.claiming`.

ΠΑΓΙΔΑ

Η transaction που μετακινεί ready σε claimed προστατεύει τη queue representation. Δεν κάνει atomic την εφαρμογή του job με το acknowledgement της queue. Worker death εξακολουθεί να δημιουργεί ambiguous business outcome.

ΜΗΧΑΝΙΣΜΟΣ

Το Solid Queue είναι durable relational state machine. Για να το debug-άρεις, βρες ποια representation row υπάρχει τώρα και ποιος actor επιτρέπεται να την μετακινήσει μετά.

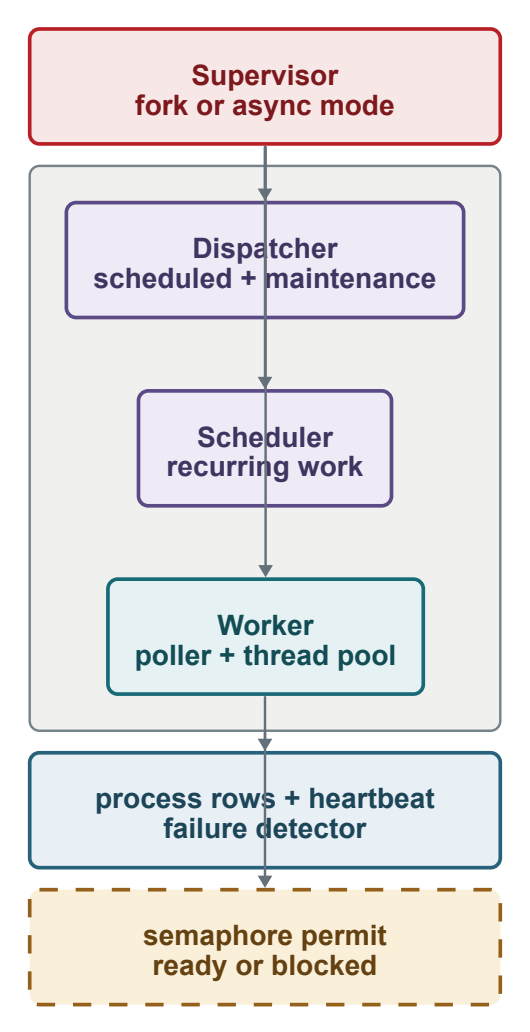
PROBE

Σε development με Solid Queue, κάνε enqueue ένα immediate, ένα scheduled και δύο concurrency-limited jobs. Επιθεώρησε μόνο IDs και timestamps στους execution tables καθώς dispatcher και worker τρέχουν. Σταμάτα worker μετά το claim και παρατήρησε πώς γίνεται visible το orphaned outcome.

ΚΕΦΑΛΑΙΟ 20

Supervisors, heartbeats και semaphores

To queue state machine χρειάζεται ζωντανούς actors. Supervisor, dispatcher, scheduler και worker έχουν διαφορετική δουλειά, διαφορετικά threads και διαφορετικό database budget.



Σχήμα 20.1 · Η process topology και το semaphore path ενός limited job.

Ο supervisor διαβάζει configuration και ξεκινά worker, dispatcher και scheduler actors. Σε default fork mode κάθε configured actor γίνεται ξεχωριστή process. Σε async mode τρέχουν ως threads στην supervisor process και το `processes` setting αγνοείται. Η έκδοση 1.5.0 συνιστά fork mode για isolation. Η επιλογή επηρεάζει memory, failure blast radius, GVL contention και συνολικά database connections.

Ο worker έχει polling loop και bounded `Concurrent::ThreadPoolExecutor`. Ζητά ready jobs ίσα με τα idle worker threads και τα κάνει post στο pool. Κάθε execution τυλίγεται στο Rails application executor. Εκτός από τα configured job threads, υπάρχουν polling και heartbeat threads. Η Solid Queue configuration μάλιστα προειδοποιεί όταν το database pool είναι μικρότερο από το εκτιμώμενο thread demand.

Heartbeat ως failure detector

Κάθε registrable process γράφει row με kind, name, pid, hostname, supervisor και metadata. Periodic timer ενημερώνει `last_heartbeat_at`. Supervisor maintenance θεωρεί παλιά rows prunable βάσει threshold. Αυτό είναι failure detector με timeout. Δεν ξεχωρίζει τέλεια dead process από process που έχει σταματήσει να scheduler-άρεται, έχασε database connectivity ή πάγωσε σε μεγάλη VM pause.

Το threshold είναι tradeoff. Πολύ μικρό προκαλεί false death και πιθανή επανεκτέλεση ενώ ο παλιός worker ζει. Πολύ μεγάλο αφήνει orphaned claims αόρατα περισσότερο. Χρειάζεται να ξεπερνά φυσιολογικά heartbeat delays αλλά να χωρά στο recovery objective.

Semaphore ως database counter

Το concurrency key αντιστοιχεί σε semaphore row με `value` και `expires_at`. Για limit `N`, η αρχική απόκτηση δημιουργεί row με `N - 1` ή μειώνει atomically θετική value. Αν δεν υπάρχει permit, το job γίνεται blocked. Όταν job τελειώνει, το signal αυξάνει value ως το limit και απελευθερώνει ένα ordered blocked job.

```ruby
class ChargeReservationJob < ApplicationJob
  limits_concurrency to: 1,
    key: ->(reservation_id) { "reservation:#{reservation_id}" },
    duration: 10.minutes

  def perform(reservation_id)
    Payments::Charge.call(reservation_id: reservation_id)
  end
end
```

Το key πρέπει να είναι stable, bounded σε μέγεθος και χωρίς sensitive data. Το duration δεν είναι maximum job runtime. Είναι recovery lease. Αν λήξει ενώ το πρώτο job ακόμη τρέχει, maintenance μπορεί να επιτρέψει άλλο job με ίδιο key. Άρα το protected operation παραμένει idempotent και, όπου χρειάζεται, κρατά database invariant ή provider idempotency key.

Concurrency limit δεν είναι ordering guarantee. Δύο blocked jobs μπορεί να γίνουν eligible με σειρά που επηρεάζεται από timestamps, locks και concurrent maintenance. Αν business contract απαιτεί sequence numbers, ο consumer πρέπει να ελέγξει expected version και να αναβάλει ή να απορρίψει out-of-order intent.

Dispatcher και scheduler

Ο dispatcher μετακινεί due scheduled executions και τρέχει concurrency maintenance που ελέγχει expired blocks. Ο scheduler παράγει recurring jobs. Αν έχεις workers χωρίς dispatcher, immediate ready jobs μπορεί να τρέχουν αλλά scheduled jobs μένουν πίσω. Αν scheduler σταματήσει, recurring work δεν δημιουργείται. Health check πρέπει να βλέπει actor-specific heartbeat και state age, όχι μόνο ότι «τρέχει bin/jobs».

Graceful termination

Ο supervisor διανέμει signals, περιμένει configured shutdown timeout και μετά μπορεί να προχωρήσει σε immediate termination. Worker σταματά να παίρνει νέα δουλειά και περιμένει thread pool. Ένα job μεγαλύτερο από timeout πρέπει να είναι safe να διακοπεί ή να χρησιμοποιεί continuation checkpoints. Η αύξηση timeout καθυστερεί deploy termination και δεν αφαιρεί host kill ή crash.

Heartbeats είναι υποψίες, όχι αποδείξεις

Ένα stale heartbeat μπορεί να σημαίνει νεκρή process, scheduler pause, database partition ή host τόσο φορτωμένο που δεν πρόλαβε να ανανεώσει. Ο failure detector δεν γνωρίζει ποια από αυτές συνέβη. Όταν cleanup χαρακτηρίσει process ως dead, η παλιά process μπορεί ακόμη να επιστρέψει. Αυτό δημιουργεί overlap window που κανένα heartbeat tuning δεν μηδενίζει.

Γι’ αυτό η ανάκτηση claims και η απελευθέρωση semaphores πρέπει να θεωρούν τον παλιό worker πιθανό late writer. Για κρίσιμο effect χρησιμοποίησε database constraint, effect ledger ή fencing token που απορρίπτει stale generation. Μεγαλύτερο heartbeat interval μειώνει false suspicions αλλά καθυστερεί recovery. Μικρότερο interval κάνει το αντίστροφο και αυξάνει database writes. Διάλεξέ το με μετρημένα pause times και explicit business tolerance, όχι με αισθητική προτίμηση.

Polling shape και database plan

Ο worker δεν ρωτά αφηρημένα «δώσε μου δουλειά». Χτίζει polling query από queue names, priority, availability και batch size. Exact queue names επιτρέπουν καθαρότερα predicates. Wildcards ή πολύ μεγάλα queue sets μπορούν να αλλάξουν το query plan και το index path. Σε μεγάλη εγκατάσταση, το queue naming scheme είναι database design surface.

Μέτρα poll duration, rows returned, empty polls και claim conflicts ανά worker configuration. Πολύ μικρό polling interval μειώνει wake-up latency αλλά αυξάνει idle database traffic. Πολύ μεγάλο interval αφήνει ready work να περιμένει. Dispatcher batch size έχει παρόμοιο tradeoff για scheduled rows: μεγάλα batches κάνουν throughput, αλλά κρατούν locks και transaction work περισσότερο.

Concurrency key migration

Το concurrency key είναι persisted coordination identity. Αν αλλάξει format σε rolling release, παλιοί και νέοι workers μπορούν να αποκτήσουν διαφορετικό semaphore για το ίδιο business resource και να τρέξουν μαζί. Κάνε dual-key transition ή drain το σχετικό work πριν από την αλλαγή. Το ίδιο ισχύει όταν ένα key βασίζεται σε mutable attribute. Προτίμησε stable aggregate id και versioned namespace.

Η semaphore duration είναι recovery clock. Το job timeout είναι execution clock. Το business deadline είναι τρίτο clock. Βάλ’ τα σε έναν πίνακα και δοκίμασε τι γίνεται όταν λήγουν με κάθε δυνατή σειρά.

SOURCE TRACE

Στο `solid_queue v1.5.0` ακολούθησε `Supervisor.start`, `Configuration#configured_processes`, `Processes::Registrable`, `Worker#poll` και `Dispatcher#poll`. Για concurrency ακολούθησε `Job::ConcurrencyControls -> Semaphore::Proxy -> BlockedExecution`. Το worker pool της έκδοσης χρησιμοποιεί `Concurrent::ThreadPoolExecutor`, όχι fiber execution mode.

ΠΑΓΙΔΑ

Μην ρυθμίσεις concurrency duration σαν εκτίμηση median runtime. Πρέπει να καλύπτει ασφαλές lease horizon και η operation πρέπει ακόμη να αντέχει overlap μετά από pause, partition ή πολύ αργό execution.

ΜΗΧΑΝΙΣΜΟΣ

Supervisor topology δίνει liveness. Semaphore δίνει bounded eligibility. Idempotency και database constraints δίνουν business safety. Είναι διαφορετικές ευθύνες.

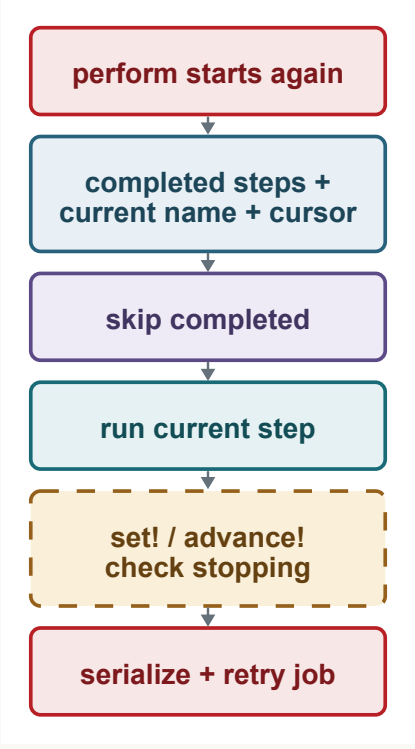
PROBE

Τρέξε δύο jobs με ίδιο key και barrier στο πρώτο. Παρατήρησε semaphore value και blocked row. Μετά προσομοίωσε duration expiry πριν φύγει το barrier. Επιβεβαίωσε αν μπορεί να υπάρξει overlap και αν το domain invariant επιβιώνει. Επανάλαβε graceful shutdown με job μικρότερο και μεγαλύτερο από timeout.

ΚΕΦΑΛΑΙΟ 21

Continuations ως durable state machine

Το continuation δεν αποθηκεύει Ruby stack. Αποθηκεύει completed step names και cursor, μετά ξανατρέχει τη `perform` από την αρχή με skip rules.



Σχήμα 21.1 · Το serialized continuation state οδηγεί skip, resume και checkpoint.

Με `ActiveJob::Continuable`, το job αποκτά `Continuation` object, resumptions counter και around-perform callback. Η `perform` εξακολουθεί να είναι κανονική method. Κάθε `step` εκτελείται μόλις συναντηθεί. Όταν ολοκληρωθεί, το name μπαίνει στα completed. Όταν cursor αλλάξει, το current step και cursor γίνονται μέρος του serializable progress. Σε resume, η job method αρχίζει ξανά και τα ήδη completed steps παραλείπονται.

Αυτό σημαίνει ότι κώδικας έξω από steps τρέχει σε κάθε execution. Πρέπει να φορτώνει current state ή να προετοιμάζει cheap context, όχι να παράγει μη-idempotent effect.

```ruby
class RebuildEventLedgerJob < ApplicationJob
  include ActiveJob::Continuable

  def perform(event_id)
    @event = Event.find(event_id)

    step :reservations, start: 0 do |step|
      @event.reservations.where("id >= ?", step.cursor).find_each do |record|
        LedgerProjector.call(record)
        step.advance!(from: record.id)
      end
    end

    step :publish, isolated: true do
      @event.update!(ledger_ready: true)
    end
  end
end
```

Cursor semantics

Το `advance!(from: record.id)` καλεί `succ`, άρα αποθηκεύει το επόμενο expected position. Με `find_each(start:)`, πρέπει να ελέγξεις inclusive semantics για τη δική σου cursor strategy. Αν αποθηκεύσεις current id αντί για successor, μπορεί να ξαναδείς row. Αυτό είναι ασφαλές μόνο αν processing είναι idempotent. Αν αποθηκεύσεις successor πριν ολοκληρωθεί το effect, μπορεί να χάσεις row.

Υπάρχει αναπόφευκτο window ανάμεσα σε business effect και durable re-enqueue του updated continuation payload. Το cursor update δεν γράφει αμέσως ξεχωριστό checkpoint row. Ελέγχει αν ο adapter σταματά και, όταν γίνει interrupt ή retried error μετά από progress, το job επανεγγράφεται με serialized continuation state. Για crash χωρίς controlled retry, η queue και application idempotency πρέπει να καλύψουν replay.

Step validation προστατεύει το πρόγραμμα

Τα step names πρέπει να είναι symbols, να εμφανίζονται μία φορά, να μην είναι nested και να εμφανίζονται με την αναμενόμενη σειρά κατά resume. Αν νέο release αλλάξει σειρά ή αφαιρέσει current step ενώ υπάρχουν suspended jobs, το continuation σηκώνει `InvalidStepError`. Άρα η step list είναι και αυτή versioned job protocol.

Για ασφαλή evolution, πρόσθεσε νέο step μετά τα παλιά ή κράτα compatibility branch βάσει explicit job contract version. Rename step ισοδυναμεί με schema rename σε persisted workflow state. Το `isolated: true` αναγκάζει νέα execution πριν από αργό step όταν έχει ήδη γίνει progress, ώστε η πρόοδος να serialized πριν μπει σε μεγάλο non-checkpointable block. Δεν κάνει το step transactional.

Checkpoint frequency και transactions

Checkpoint ελέγχει `queue_adapter.stopping?`. Αν όχι, συνεχίζει χωρίς enqueue. Αν ναι, σηκώνει `Continuation::Interrupt`, που κληρονομεί από `Exception` ώστε να μην καταποθεί από συνηθισμένο `rescue StandardError`. Το around callback καλεί `retry_job` με continuation state.

Η συχνότητα πρέπει να είναι μικρότερη από graceful shutdown window. Πολύ συχνά checkpoints προσθέτουν control overhead και απαιτούν πολύ λεπτά idempotency boundaries. Πολύ αραιά δεν βοηθούν deploy termination. Μην checkpoint-άρεις μέσα σε ανοιχτή business transaction. Η queue retry και η database commit δεν είναι ένα atomic state transition.

Resume after ordinary errors

Από default, αν ordinary `StandardError` συμβεί αφού το continuation έκανε progress, το module προσπαθεί resume μέσω `retry_job`. Αυτό αποφεύγει να χαθεί known progress, αλλά μπορεί να αλλάξει το retry policy που νόμιζες ότι ελέγχει μόνο `retry_on`. Κατέγραψε `resumptions`, Active Job executions και domain attempts χωριστά. Το optional max resumptions είναι safety cap, όχι business completion guarantee.

Step evolution ανάμεσα σε releases

Το serialized continuation state είναι deployed protocol. Job που ξεκίνησε με release A μπορεί να συνεχίσει αφού το release B έχει αλλάξει step names ή order. Η validation που απορρίπτει αδύνατη μετάβαση είναι πολύτιμη, αλλά το production ζητούμενο είναι να μην παράγεις αδύνατη μετάβαση. Κράτα παλιό step path ώσπου να στραγγίξουν τα payloads ή γράψε explicit state migration που είναι ασφαλές να ξανατρέξει.

Ένα step πρέπει να έχει μικρό, versioned input contract. Αν στηρίζεται σε current scope, feature flag ή query ordering που άλλαξε, το ίδιο cursor μπορεί να σημαίνει άλλο dataset μετά το deploy. Για μεγάλα projections κράτα stable ordering με unique tiebreaker και, όπου απαιτείται, snapshot upper bound. Τότε το cursor σημαίνει «όλα μέχρι αυτό το key μέσα σε αυτό το snapshot», όχι απλώς έναν ασαφή αριθμό που κάποτε φάνηκε να δουλεύει.

Advance και effect boundary

Το πιο επικίνδυνο σημείο είναι η σειρά `effect -> advance -> serialized retry`. Αν η process πεθάνει μετά το effect αλλά πριν αποθηκευτεί ο νέος cursor, το effect επαναλαμβάνεται. Αν αποθηκεύσεις cursor πριν από το effect και πεθάνεις μετά, το effect χάνεται. Η continuation δεν εξαφανίζει αυτό το δίλημμα. Το λύνεις με idempotent effect, local transaction που γράφει effect ledger μαζί με progress ή outbox για το remote βήμα.

Το `advance!` ενημερώνει control state στο τρέχον job object. Η durability έρχεται όταν το ενημερωμένο payload γίνει enqueue για resume. Σε ordinary execution που συνεχίζει χωρίς interruption, πολλά advances μπορεί να υπάρχουν μόνο στη μνήμη μέχρι να χρειαστεί checkpoint. Άρα το cursor είναι recovery marker με συγκεκριμένα persistence moments, όχι write-ahead log κάθε iteration.

Progress χωρίς ακριβό recount

Μην υπολογίζεις progress με full `COUNT` σε κάθε batch αν το dataset είναι μεγάλο ή αλλάζει. Κράτα processed counter ως hint και cursor ως correctness state. Το UI μπορεί να δείχνει approximate progress, αλλά completion πρέπει να προκύπτει από exhausted stable query ή explicit terminal step. Αν νέα rows μπαίνουν συνεχώς, όρισε upper bound στην αρχή ή αποδέξου ότι το job είναι daemon-like reconciler και όχι πεπερασμένο continuation.

SOURCE TRACE

Η συμπεριφορά είναι στα `active_job/continuable.rb`, `continuation.rb` και `continuation/step.rb`. Ακολούθησε `around_perform :continue -> step -> advance! -> checkpoint! -> interrupt! -> retry_job -> serialize`.

ΠΑΓΙΔΑ

Το cursor είναι δική σου εγγύηση. Το Rails δεν ξέρει αν δείχνει last processed, next unprocessed ή snapshot boundary. Γράψε αυτή τη σημασία δίπλα στον step και δοκίμασε crash πριν και μετά από κάθε advance.

ΜΗΧΑΝΙΣΜΟΣ

Continuation σημαίνει replayable program με μικρό serialized control state. Δεν παγώνει stack, local variables ή database snapshot. Κάθε execution ξαναχτίζει τον κόσμο γύρω από το cursor.

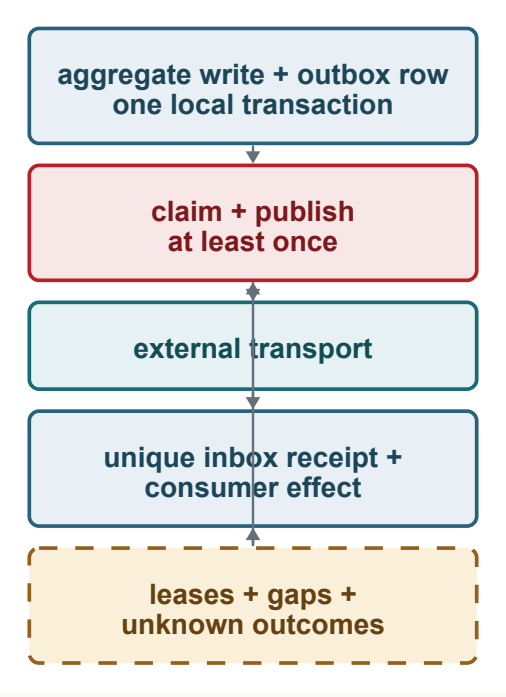
PROBE

Με fake adapter stopping signal, διέκοψε το job μετά από συγκεκριμένο record. Επιθεώρησε το serialized `continuation` hash και κάνε resume με νέο job instance. Μετά άλλαξε προσωρινά step order για να δεις το validation failure. Τέλος κάνε το projector idempotent και προσομοίωσε hard crash ακριβώς πριν και μετά το `advance!`.

ΚΕΦΑΛΑΙΟ 22

Outbox, inbox και reconciliation

Όταν δύο systems δεν μοιράζονται commit, η ασφάλεια έρχεται από durable intent, stable identity και επαναλαμβανόμενο reconciliation.



Σχήμα 22.1 · Η outbox γεφυρώνει το local commit με at-least-once delivery και inbox dedupe.

Το transactional outbox δεν είναι Rails feature. Είναι pattern που χρησιμοποιεί το Active Record transaction boundary. Η ίδια transaction που αλλάζει το aggregate γράφει row με event identity, type, version, aggregate identity και payload ή reference. Μετά από commit, dispatcher διαβάζει unpublished rows και τα παραδίδει. Αν πέσει, τα ξαναβλέπει.

```ruby
Reservation.transaction do
  reservation = Reservation.lock.find(reservation_id)
  reservation.confirm!

  OutboxEvent.create!(
    event_id: SecureRandom.uuid,
    event_type: "reservation.confirmed",
    event_version: 2,
    aggregate_type: "Reservation",
    aggregate_id: reservation.id,
    payload: { event_id: reservation.event_id }
  )
end
```

Unique index στο `event_id` δίνει stable message identity. Αν ίδιο business intent μπορεί να δημιουργηθεί ξανά, χρειάζεται επιπλέον unique business key, όπως `(reservation_id, event_type, confirmation_version)`. Ένα random UUID σε κάθε retry δεν κάνει dedupe.

Claim, publish, mark

Ο dispatcher συνήθως claim-άρει batch με row locks, στέλνει και μετά σημειώνει published. Δεν μπορεί να κρατήσει μία database transaction ανοιχτή γύρω από unbounded network publish χωρίς μεγάλο lock lifetime. Αν κάνει commit το claim πριν το publish, process death αφήνει claimed row που χρειάζεται lease expiry. Αν publish και mark είναι χωριστά, υπάρχει window όπου publish πέτυχε αλλά mark δεν γράφτηκε. Το event θα σταλεί ξανά.

Αυτό δεν είναι bug που λείπει ένα callback για να διορθωθεί. Είναι το θεμελιώδες at-least-once boundary. Ο downstream consumer χρειάζεται idempotency.

```output
local commit -> claim -> external publish -> local published_at
                                |
                                + crash here means duplicate on retry
```

Σε PostgreSQL, `FOR UPDATE SKIP LOCKED` επιτρέπει πολλούς dispatchers να παίρνουν διαφορετικές rows. Σε άλλο adapter χρησιμοποίησε το αντίστοιχο supported locking contract. Ordering ανά aggregate χρειάζεται sequence και consumer check, επειδή parallel claim μπορεί να αναδιατάξει delivery.

Inbox στον consumer

Ο consumer γράφει inbox receipt με unique `(consumer, event_id)` στην ίδια local transaction με το effect του. Αν το insert συγκρουστεί, το event έχει ήδη εφαρμοστεί και μπορεί να acknowledged. Αν το effect αφορά εξωτερικό provider, η inbox row από μόνη της δεν κάνει το provider call atomic. Χρειάζεται local intent και δεύτερος worker ή provider idempotency.

```ruby
InboxReceipt.transaction do
  receipt = InboxReceipt.create_or_find_by!(
    consumer: "seat_allocator",
    event_id: event_id
  )
  next if receipt.applied_at?

  SeatProjection.apply!(payload)
  receipt.update!(applied_at: Time.current)
end
```

Το `next` μέσα στο block είναι valid Ruby control flow προς το τέλος του block. Το database constraint, όχι το initial SELECT, κρατά τη μοναδικότητα υπό race.

Reconciliation είναι κανονικό execution path

Ο reconciler ψάχνει stuck rows, όχι μόνο exceptions. Unpublished outbox older than threshold, claimed lease που έληξε, inbox receipt χωρίς applied timestamp, provider operation με unknown local outcome και aggregate version gap. Η δουλειά του είναι να επαναφέρει state machine σε γνωστή πορεία, όχι να «καθαρίζει σκουπίδια».

Κάθε state χρειάζεται owner, deadline και next action. Ένα generic `failed_at` δεν αρκεί αν δεν ξέρεις αν επιτρέπεται retry. Κράτα last error category, attempt count και next attempt time. Μην αποθηκεύεις ολόκληρα exception messages αν περιέχουν secrets ή unbounded cardinality.

Payload versus re-read

Snapshot payload δίνει event-time facts αλλά μεγαλώνει compatibility schema. Reference payload φορτώνει current state και μπορεί να χάσει την ιστορική μετάβαση. Για `reservation.confirmed`, το downstream ticket χρειάζεται συχνά immutable price, currency και attendee identity της επιβεβαίωσης. Αυτά ανήκουν στο versioned event, όχι σε later read ενός mutable reservation.

Leases, poison events και sequence gaps

Το claim ενός outbox row είναι lease, όχι ιδιοκτησία για πάντα. Αποθήκευσε claim generation ή token και κάνε conditional mark-published, ώστε worker που επιστρέφει αργά μετά από lease expiry να μην πατήσει νεότερη απόφαση. Το `SKIP LOCKED` μοιράζει διαθέσιμη δουλειά. Δεν λύνει μόνο του το late-writer problem μετά την transaction που έκανε claim.

Ένα event που αποτυγχάνει μόνιμα δεν πρέπει να μπλοκάρει για πάντα όλο το aggregate. Χρειάζεται explicit poison state, bounded retry και operator action. Αν consumers απαιτούν per-aggregate ordering, το skip δημιουργεί sequence gap που πρέπει να φαίνεται ως state, όχι να κρύβεται. Ο consumer μπορεί να παγώσει μόνο το συγκεκριμένο aggregate, να ζητήσει snapshot repair ή να εφαρμόσει νεότερο event μόνο όταν το protocol το επιτρέπει. Η επιλογή είναι business semantics. Ο dispatcher δεν μπορεί να τη συμπεράνει από ένα exception.

SOURCE TRACE

Δεν υπάρχει `ActiveRecord::Outbox`. Ο local atomicity μηχανισμός είναι το `Active Record transaction stack`. Για database-backed dispatch μπορείς να μελετήσεις το claim pattern του `SolidQueue::ReadyExecution`, χωρίς να εξαρτήσεις application schema από private Solid Queue models.

ΠΑΓΙΔΑ

Το `published_at` σημαίνει «ο publisher πήρε επιτυχία και το κατέγραψε». Δεν αποδεικνύει ότι κάθε consumer εφάρμοσε το event. End-to-end completion χρειάζεται consumer receipt ή domain-specific reconciliation.

ΜΗΧΑΝΙΣΜΟΣ

Outbox προστατεύει local intent. Inbox προστατεύει local application του message. Reconciliation προστατεύει τα ambiguous και stuck states ανάμεσά τους.

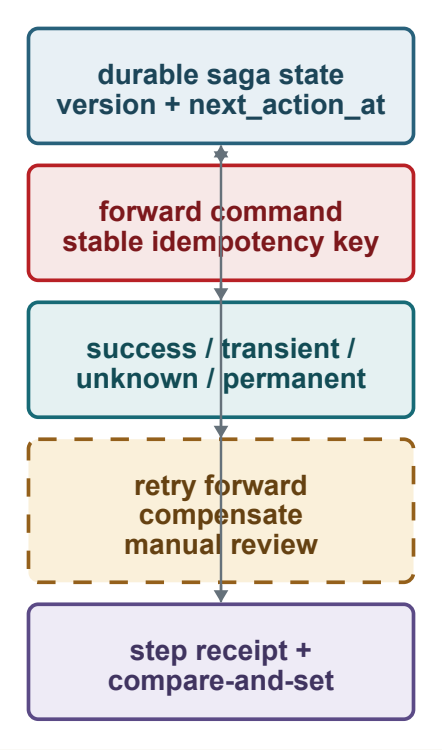
PROBE

Βάλε deterministic crash injection μετά το local commit, μετά το claim, μετά το external publish και πριν το inbox commit. Τρέξε dispatcher και consumer ξανά. Assert μία τελική business αλλαγή, επιτρεπτά duplicate deliveries και μηδενικά stuck rows μετά τον reconciler.

ΚΕΦΑΛΑΙΟ 23

Sagas και compensations

Όταν ένα use case περνά πολλά durable systems, δεν υπάρχει μαγικό rollback. Υπάρχει state machine με forward steps, compensations και ανθρώπινη έξοδο.



Σχήμα 23.1 · Μια saga προχωρά, αποτυγχάνει και επιλέγει retry, compensation ή review.

Η κράτηση μπορεί να δεσμεύσει θέση, να εγκρίνει πληρωμή και να εκδώσει εισιτήριο. Καμία database transaction του Seatly δεν περιλαμβάνει atomic commit με payment provider και email system. Μια saga αποθηκεύει το workflow state και εκτελεί κάθε remote step ως idempotent command. Αν step αποτύχει, retry-άρει, αντισταθμίζει ή περνά σε manual review.

Compensation δεν γυρίζει τον χρόνο. Refund μπορεί να αποτύχει, να έχει fee ή να εμφανιστεί αργότερα στον χρήστη. Release θέσης μπορεί να μην επιτρέπεται αφού άρχισε event. Γι' αυτό η saga πρέπει να ονομάζει business states όπως `payment_unknown`, `refund_pending` και `manual_review`, όχι μόνο `failed`.

Durable orchestration

Στο orchestration model μία row κρατά current state, version και step attempts. Ένα job φορτώνει με lock, αποφασίζει next command, γράφει durable intent και καλεί ή προγραμματίζει τον adapter. Η remote κλήση δεν μένει μέσα σε μακριά database transaction. Το result εφαρμόζεται με compare-and-set πάνω στη saga version ώστε stale response να μην προχωρήσει λάθος generation.

```ruby
class BookingSaga < ApplicationRecord
  enum :state, {
    reserving: 0,
    charging: 1,
    issuing: 2,
    compensating: 3,
    completed: 4,
    manual_review: 5
  }
end
```

Το enum δεν είναι η state machine. Constraints, transition operation και idempotency keys είναι η state machine. Ένα generic `update!(state: ...)` από οποιοδήποτε caller παρακάμπτει expected-current-state check.

```ruby
changed = BookingSaga.where(id: saga_id, state: :charging, version: version)
  .update_all(state: BookingSaga.states.fetch(:issuing), version: version + 1)

raise BookingSaga::StaleTransition unless changed == 1
```

Η remote response κουβαλά command id και expected saga version. Late success από attempt 1 δεν πρέπει να αντικαταστήσει compensation που ξεκίνησε μετά από attempt 2 timeout.

Choreography και event loops

Στο choreography model κάθε component αντιδρά σε events και παράγει άλλα. Μειώνει central coordinator αλλά κάνει τη συνολική πορεία δυσκολότερη να παρατηρηθεί. Cycles και duplicate reactions μπορούν να εμφανιστούν χωρίς explicit workflow row. Για μικρό Rails monolith, durable orchestrator είναι συχνά πιο καθαρός. Choreography αξίζει όταν bounded contexts είναι πραγματικά ανεξάρτητα και τα event contracts ήδη υπάρχουν.

Forward recovery πρώτα

Πριν προσθέσεις compensation, ρώτησε αν safe retry μπορεί να ολοκληρώσει το forward step. Ένα προσωρινό ticket service outage δεν απαιτεί refund αμέσως. Μπορεί να χρειάζεται `issuing` με deadline. Μετά το deadline, policy αποφασίζει refund ή review. Αυτό αποφεύγει oscillation όπου transient error προκαλεί ακριβή αντιστάθμιση που μετά πρέπει να αναιρεθεί.

Τα deadlines είναι data. Μην βασίζεσαι μόνο σε scheduled job, επειδή enqueue μπορεί να χαθεί ή να καθυστερήσει. Reconciler βρίσκει saga rows των οποίων `next_action_at` πέρασε και ξαναπρογραμματίζει.

Compensation stack και partial order

Δεν αντιστρέφονται όλα τα steps με απλή reverse σειρά. Αν δύο steps ήταν ανεξάρτητα, οι compensations μπορούν να τρέξουν παράλληλα. Αν ticket issuance εξαρτάται από charge, void ticket μπορεί να πρέπει να προηγηθεί refund. Γράψε dependency graph και αποθήκευσε completion receipts ανά step.

Κάθε compensation έχει δικό της idempotency identity. `refund:payment_123` είναι διαφορετική command από `charge:reservation_42`. Provider status query προηγείται blind retry όταν το προηγούμενο outcome είναι unknown.

Τι βλέπει ο χρήστης

Distributed workflow μπορεί να είναι pending χωρίς να είναι broken. Το UI πρέπει να δείχνει stable public state και recovery expectation. Μην παρουσιάσεις «απέτυχε» αν το system ακόμη retry-άρει έκδοση εισιτηρίου. Μην παρουσιάσεις «ολοκληρώθηκε» μόνο επειδή η local transaction έκανε commit.

SOURCE TRACE

Το Rails δεν παρέχει saga coordinator. Τα primitives είναι Active Record transactions, optimistic updates και Active Job retries. Μελέτησε το `Persistence` update constraint path, το `Active Job retry path` και το προηγούμενο outbox boundary. Η orchestration policy παραμένει application code και schema.

ΠΑΓΙΔΑ

Μην ονομάζεις compensation μια δεύτερη destructive πράξη χωρίς independent idempotency και reconciliation. Η compensation μπορεί να αποτύχει και χρειάζεται πλήρες production lifecycle.

ΜΗΧΑΝΙΣΜΟΣ

Saga correctness δεν είναι «όλα ή τίποτα». Είναι κάθε durable state να έχει σαφή next action, safe repetition, deadline και terminal path που μπορεί να εξηγηθεί σε άνθρωπο.

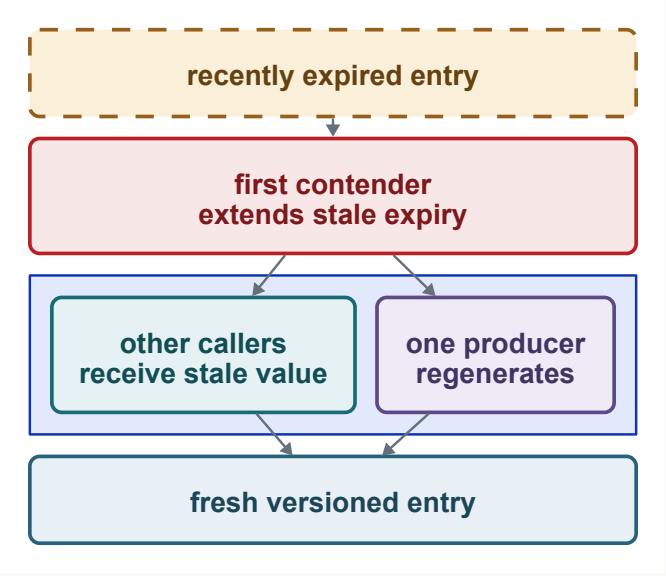
PROBE

Φτιάξε model-based test του workflow. Παρήγαγε sequences από duplicate success, timeout, late response, worker crash και compensation failure. Μετά από κάθε sequence assert legal state, μοναδικά provider idempotency keys και ότι κάθε nonterminal state έχει `next_action_at` ή manual owner.

ΚΕΦΑΛΑΙΟ 24

Cache stampedes, versions και race windows

To cache stampede δεν είναι απλώς πολλά misses. Είναι συγχρονισμένη παραγωγή ακριβού value ακριβώς όταν το dependency είναι ήδη πιεσμένο.



Σχήμα 24.1 · Το race_condition_ttl δίνει stale value ενώ ένας producer ανανεώνει.

Το `Rails.cache.fetch(name, options)` κανονικοποιεί key, διαβάζει entry, ελέγχει expiration και version και είτε επιστρέφει value είτε τρέχει το block. Το store αποθηκεύει `ActiveSupport::Cache::Entry` με value, expiry και optional version. Το key και το version είναι διαφορετικά: το normalized key βρίσκει entry, ενώ version mismatch το κάνει logical miss χωρίς απαραίτητα νέο physical key.

race_condition_ttl ως stale-while-revalidate

Αν expired entry είναι αρκετά πρόσφατο και υπάρχει positive `race_condition_ttl`, ο πρώτος contender επεκτείνει προσωρινά την expiry και ξεκινά regeneration. Άλλοι callers διαβάζουν το παλιό value στο μικρό window. Όταν ο producer γράψει νέο value, το stale αντικαθίσταται. Αν producer αποτύχει, μετά το extension άλλος μπορεί να δοκιμάσει.

```ruby
summary = Rails.cache.fetch(
  ["event-summary", event.cache_key_with_version],
  expires_in: 5.minutes,
  race_condition_ttl: 10.seconds
) do
  EventSummary.build(event.id)
end
```

Το feature επιτρέπεται μόνο αν stale value για έως το window είναι ασφαλές. Δεν ταιριάζει σε remaining seats που χρησιμοποιούνται για authorization ή purchase invariant. Ταιριάζει περισσότερο σε display summary που έχει explicit freshness tolerance.

Η implementation γράφει προσωρινά expired entry πίσω με μεγαλύτερο physical expiry. Τα ακριβή atomicity properties εξαρτώνται από store. Δύο processes μπορεί ακόμη να regenerate υπό races, partition ή backend που δεν δίνει τις ίδιες conditional guarantees. Το block παραμένει safe να τρέξει περισσότερες φορές.

Local cache αλλάζει το observation

Stores που περιλαμβάνουν `Strategy::LocalCache` μπορούν να κρατούν serialized entries σε execution-local memory για τη διάρκεια request-like block. Repeated read αποφεύγει network call. Write και delete ενημερώνουν ή καθαρίζουν local entry. Αυτό βελτιώνει latency αλλά σημαίνει ότι backend metrics δεν βλέπουν κάθε logical read. Το local cache registry βρίσκεται στο `IsolatedExecutionState`.

Μη μπερδέψεις local cache με process-wide MemoryStore. Το πρώτο έχει bounded execution lifetime. Το δεύτερο έχει ξεχωριστό content ανά process και μπορεί να δίνει διαφορετικές τιμές πίσω από load balancer.

Key cardinality και namespace evolution

Ένα key πρέπει να περιέχει όλες τις dimensions που αλλάζουν το value και καμία unbounded τυχαία dimension. Tenant, locale, permission tier, template version και record version μπορεί να είναι relevant. Request id σχεδόν ποτέ δεν είναι, επειδή ακυρώνει reuse. Sensitive values δεν ανήκουν σε observable cache keys.

Για rolling release, νέο code που δεν μπορεί να διαβάσει παλιό serialized value χρειάζεται new namespace ή tolerant decoder. Cache δεν είναι database, αλλά παλιά entries επιβιώνουν deploy. Ένα class rename μπορεί να σπάσει Marshal payload. Προτίμησε primitive versioned representations για shared caches.

Multi fetch και partial hits

Το `fetch_multi` κάνει bulk read, παράγει μόνο misses και μετά bulk write όπου το store το υποστηρίζει. Είναι σημαντικό σε collections επειδή μειώνει network round trips. Δεν εγγυάται atomic snapshot όλων των keys. Κάθε entry μπορεί να ανήκει σε διαφορετική version στιγμή. Αν χρειάζεσαι coherent aggregate, cache ένα versioned aggregate ή βάλε generation token σε όλα τα components.

Metrics που δεν προκαλούν νέο πρόβλημα

Τα `cache_read`, `cache_fetch_hit`, `cache_generate`, `cache_write` events δίνουν operation, store και hit metadata. Μην κάνεις metric label το full normalized key. Παράγαγε bounded namespace, όπως το πρώτο static component. Μέτρα hit rate, generate duration, error rate, value size buckets και concurrent regenerations.

Entry format είναι compatibility surface

Το cache store δεν κρατά απαραιτήτως το Ruby object όπως το βλέπει ο caller. Υπάρχει entry metadata, expiry, coder, serializer και πιθανή compression. Αλλαγή serializer ή compressor σε rolling deploy σημαίνει ότι παλιοί και νέοι workers διαβάζουν κοινό byte protocol. Χρησιμοποίησε framework-supported format transitions ή νέο namespace και δοκίμασε bidirectional compatibility πριν από release. Ένα «καθάρισε όλο το cache» μπορεί να είναι ακριβό production event, όχι αθώα λύση.

Η version ενός cache entry δεν αντικαθιστά το key namespace. Το key βρίσκει την περιοχή αποθήκευσης. Η version κρίνει αν το περιεχόμενο είναι αποδεκτό για τον caller. Σε high-churn aggregate, version mismatch μπορεί να αφήσει πολλά νεκρά bytes μέχρι eviction. Μέτρα logical misses, physical key growth και eviction ξεχωριστά. Διαφορετικά μια φαινομενικά καλή hit rate μπορεί να κρύβει memory pressure και συνεχές churn στο backend.

SOURCE TRACE

Η fetch και race TTL logic βρίσκονται στο `active_support/cache.rb` και το entry format στο `active_support/cache/entry.rb`. Η request-local layer είναι το `strategy/local_cache.rb`. Store-specific methods καθορίζουν τις πραγματικές distributed guarantees.

ΠΑΓΙΔΑ

Το stale-while-revalidate είναι product consistency policy. Μην ενεργοποιείς `race_condition_ttl` μόνο για να πέσει το load χωρίς να γράψεις πόσο stale data αντέχει ο caller.

ΜΗΧΑΝΙΣΜΟΣ

Ένα cache entry χρειάζεται identity, version, freshness budget και regeneration policy. Αν λείπει ένα από αυτά, το cache behavior θα αποφασιστεί τυχαία σε peak traffic.

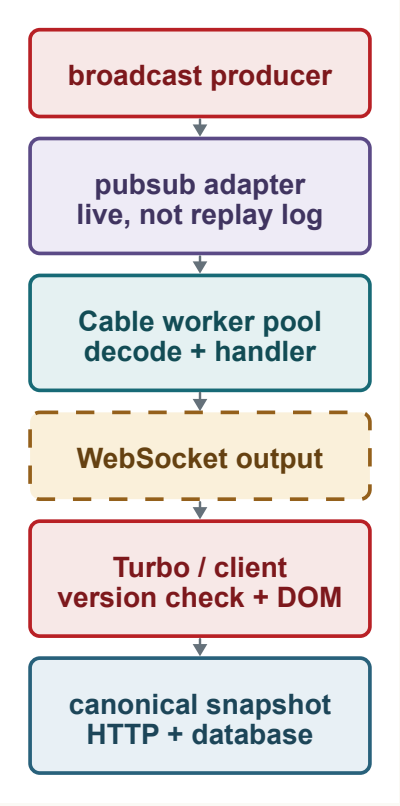
PROBE

Με barrier, κάνε 20 threads ή processes να ζητήσουν το ίδιο entry ακριβώς μετά την expiry. Μέτρησε πόσες φορές έτρεξε generator, πόσοι πήραν stale value και πόσο φορτίστηκε το dependency. Επανάλαβε με producer exception και με version mismatch. Χρησιμοποίησε τον production store adapter σε isolated environment.

ΚΕΦΑΛΑΙΟ 25

Cable και Turbo υπό ordering και backpressure

To realtime transport μεταφέρει hints γρήγορα. Δεν αντικαθιστά durable state, replay log ή client-side version check.



Σχήμα 25.1 · Το broadcast περνά pubsub, worker pool και WebSocket προς versioned DOM state.

Το `ActionCable.server.broadcast` κωδικοποιεί message και το στέλνει στον configured pubsub adapter. Ένα channel `stream_from` κάνει subscribe σε named broadcasting. Η pubsub callback δεν τρέχει user code πάνω στο event loop. Το Action Cable τη μεταφέρει στο server worker pool. Εκεί decoder ή custom handler τρέχει και τελικά καλεί `transmit` προς τη connection.

Αυτή η διαδρομή έχει τουλάχιστον τρεις queues: pubsub, worker executor και socket output. Το `worker_pool_size` περιορίζει concurrent channel work και πρέπει να έχει αντίστοιχο Active Record pool budget αν handlers κάνουν queries. Αργός handler μπορεί να καθυστερήσει άσχετες connections που μοιράζονται pool.

Ordering είναι scoped και εύθραυστο

Ένας pubsub adapter μπορεί να διατηρεί σειρά ανά channel σε normal operation, αλλά parallel producers, async broadcast jobs, reconnects και worker pool scheduling δεν δίνουν global total order. Το Cable API δεν υπόσχεται durable replay. Client που αποσυνδέθηκε χάνει messages και επανασυνδέεται στη σημερινή ροή.

Για stateful UI, κάθε update χρειάζεται aggregate version. Ο client εφαρμόζει μόνο update νεότερο από αυτό που έχει ή ζητά fresh snapshot όταν βλέπει gap. Για commutative hints, όπως «refresh this frame», exact ordering μπορεί να μην χρειάζεται.

```ruby
ActionCable.server.broadcast(
  "event:42",
  type: "reservation_count_changed",
  event_id: 42,
  version: event.lock_version
)
```

Το client δεν εμπιστεύεται τον count μέσα σε παλιό message. Μπορεί να ζητήσει current representation με conditional HTTP request. Έτσι το broadcast είναι invalidation hint και η canonical state μένει στη database/API.

Turbo Streams και render timing

Το Turbo μετατρέπει broadcast σε actions όπως append, replace, update, remove ή refresh με DOM targets. Τα `_later` APIs enqueue Active Job. Το template render γίνεται όταν τρέξει το job, άρα μπορεί να δει νεότερο record state από τη στιγμή του original commit. Αυτό είναι συχνά καλό coalescing, αλλά δεν είναι event-time snapshot.

Page refresh broadcasts στο turbo-rails 2.0.23 χρησιμοποιούν thread debouncer και request id. Πολλαπλές refresh requests στο ίδιο μικρό context μπορούν να συγχωνευτούν. Το request id βοηθά τον initiating client να συσχετίσει refresh. Δεν δημιουργεί durable delivery ούτε ordering ανάμεσα σε διαφορετικές worker processes.

Signed stream name αποτρέπει client tampering με το subscription name. Δεν αντικαθιστά application authorization όταν το δικαίωμα εξαρτάται από current membership. Custom channel μπορεί να verify το signed name και μετά να ελέγχει current user πριν `stream_from`.

Backpressure

Αν producer εκπέμπει ταχύτερα από όσο pubsub, worker pool ή socket μπορεί να καταναλώσει, κάπου μεγαλώνει queue, αυξάνει latency ή πέφτουν messages. Το Action Cable abstraction δεν μπορεί να κάνει έναν αργό browser γρήγορο. Μείωσε granularity: μία refresh hint αντί για εκατό per-row replaces, batch updates ή coalesce ανά aggregate.

Χρειάζεσαι metrics για worker executor saturation, broadcast-to-transmit delay, connection count, reconnect rate και message size. Full stream names μπορεί να έχουν user IDs και μεγάλη cardinality. Κανονικοποίησε channel class και message type.

Snapshot plus live stream

Το robust subscribe protocol είναι: authorize, πάρε snapshot με version, κάνε subscribe και χειρίσου το μικρό race ανάμεσά τους. Μία τεχνική είναι subscribe πρώτα, buffer messages, fetch snapshot και μετά apply messages με version πάνω από snapshot. Άλλη είναι durable event cursor. Για απλό UI, ένα refresh μετά το subscribe συχνά αρκεί. Διάλεξε βάσει του πόσο data loss αντέχει η οθόνη.

Stream identity και authorization lifetime

Signed stream names προστατεύουν από client-side κατασκευή αυθαίρετου stream identifier. Δεν μετατρέπουν παλιά authorization απόφαση σε διαρκή δικαίωση. Αν χρήστης χάσει πρόσβαση ενώ το socket μένει ανοικτό, πρέπει να αποφασίσεις αν το channel κάνει periodic reauthorization, αν disconnect-άρει μέσω server event ή αν το permission ισχύει μέχρι reconnect. Για sensitive updates, γράψε το lifetime ρητά και δοκίμασε revocation με ήδη subscribed client.

Υπάρχει επίσης race στη σύνδεση. Το browser μπορεί να πάρει snapshot, να χάσει event πριν ολοκληρωθεί το subscription και μετά να φαίνεται online με stale DOM. Το Turbo refresh κάνει recovery φθηνό για πολλές οθόνες, ειδικά με morph. Δεν παρέχει όμως durable cursor. Αν ο χρήστης παίρνει οικονομική ή irreversible απόφαση από αυτή την οθόνη, το action endpoint πρέπει να ξαναελέγχει canonical version και invariant. Το realtime UI μειώνει latency αντίληψης. Δεν γίνεται authority για το write.

SOURCE TRACE

Η server path βρίσκεται στα `server/broadcasting.rb`, `channel/streams.rb` και `server/worker.rb`. Για Turbo δες το tagged `turbo-rails v2.0.23`, ειδικά `Turbo::Streams::Broadcasts` και `BroadcastJob`.

ΠΑΓΙΔΑ

Μη χρησιμοποιείς broadcast arrival ως απόδειξη ότι ο client είδε state. Το socket μπορεί να κλείσει μετά το server transmit και πριν το DOM apply. Critical workflow χρειάζεται server-side durable acknowledgement ή επόμενο canonical read.

ΜΗΧΑΝΙΣΜΟΣ

Cable είναι live transport. Turbo Streams είναι UI mutation protocol. Database και versioned HTTP representation παραμένουν το recovery path όταν delivery ή ordering σπάσει.

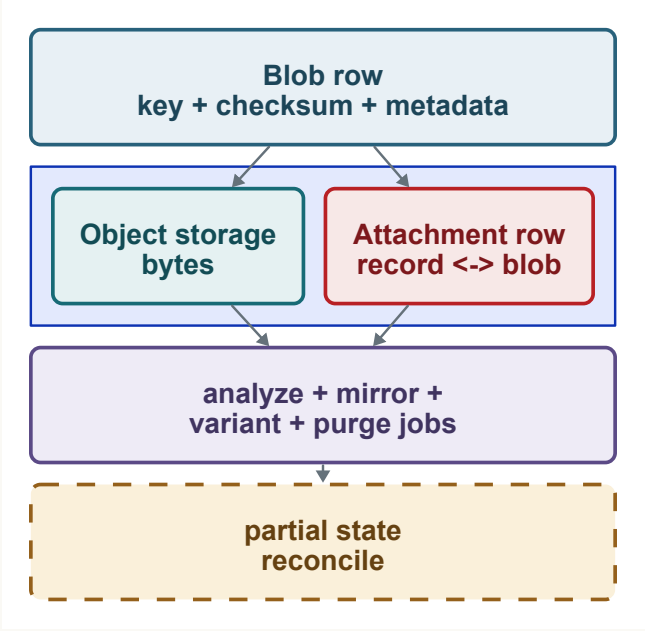
PROBE

Πρόσθεσε sequence number σε test stream. Καθυστέρησε έναν worker handler, στείλε parallel broadcasts και κάνε client reconnect ανάμεσα. Κατέγραψε duplicates, gaps και inversions. Μετά εφάρμοσε version rejection και snapshot refresh και assert ότι το τελικό DOM αντιστοιχεί στο canonical record.

ΚΕΦΑΛΑΙΟ 26

Active Storage και η συνέπεια τριών συστημάτων

Ένα attachment ζει σε database blob row, polymorphic attachment row και object storage key. Καμία κοινή transaction δεν αγκαλιάζει και τα τρία.



Σχήμα 26.1 · Τα failure windows ανάμεσα σε blob, object, attachment και purge.

Το `ActiveStorage::Blob` κρατά metadata, checksum, byte size, content type, service name και key. Το πραγματικό byte object βρίσκεται στο service. Το `ActiveStorage::Attachment` συνδέει blob με application record. Ένας blob μπορεί να έχει πολλά attachments και foreign key εμποδίζει purge ενώ συνδέεται.

Η προτεινόμενη immutability είναι σημαντική: το key αναφέρεται σε συγκεκριμένο file. Αν το content αλλάξει, δημιουργείς νέο blob. Metadata μπορεί να ενημερωθεί, αλλά η αντικατάσταση bytes κάτω από ίδιο key σπάει checksum, CDN caches, variants και consumers που θεωρούν το signed reference σταθερό.

Server-side upload

Το `create_and_upload!` κάνει unfurl του IO, αποθηκεύει blob row και μετά ανεβάζει bytes. Η row αποθηκεύεται πρώτη για να δεσμευτεί unique key. Αν process πέσει μετά το blob commit και πριν

ολοκληρωθεί upload, υπάρχει blob που δείχνει σε missing ή partial object. Database rollback δεν διαγράφει network effect που έχει ήδη φτάσει στο service.

```ruby
blob = ActiveStorage::Blob.create_and_upload!(
  io: file,
  filename: "guest-list.csv",
  content_type: "text/csv",
  identify: true
)
reservation.guest_list.attach(blob)
```

Για critical ingestion, κράτα application state `uploading`, `verifying`, `ready`, `quarantined` και `failed`. Μην θεωρείς την παρουσία blob row ως ready. Ένα verification job μπορεί να κάνει service existence, checksum ή scanner checks πριν δημοσιεύσει το file.

Direct upload

Το direct upload δημιουργεί πρώτα persisted blob χωρίς object μέσω `create_before_direct_upload!`, δίνει signed id και short-lived service URL, ο browser ανεβάζει απευθείας και αργότερα η form submission δημιουργεί attachment. User μπορεί να κλείσει tab σε κάθε βήμα. Γι' αυτό unattached blobs είναι αναμενόμενα, όχι απόδειξη corruption.

Cleanup πρέπει να κοιτά blobs αρκετά παλιά ώστε να έχει λήξει κάθε legitimate form και retry window. Το `unattached` scope έχει race με concurrent attach. Πριν purge, ξαναέλεγξε attachment υπό κατάλληλο database constraint. Η foreign key είναι τελευταία προστασία, αλλά το external delete δεν γίνεται atomic με τη recheck.

Commit callbacks και derivatives

Μετά τη δημιουργία attachment και commit, callbacks προγραμματίζουν mirror, analysis και preprocessed variant work. Άρα metadata όπως dimensions μπορεί να είναι άγνωστο προσωρινά. UI και domain code πρέπει να αντέχουν `analyzed?` false. Job retries μπορεί να τρέξουν analyzer παραπάνω από μία φορά.

Mirror service δίνει migration και availability δυνατότητες, όχι synchronous multi-object transaction. Primary upload μπορεί να υπάρχει πριν το mirror. Read policy, repair job και cutover πρέπει να βασίζονται σε observed copy status.

Purge order

Το `Blob#purge` καταστρέφει database row και μετά διαγράφει service object. Αν το delete αποτύχει μετά το destroy, μένει orphan object χωρίς blob reference. Αν foreign key δείξει attachment, το purge καταπίνει το invalid foreign key path και δεν διαγράφει object. Το `purge_later` μεταφέρει αργό service I/O σε job και είναι σωστότερο μέσα από request ή callback, αλλά εξακολουθεί να χρειάζεται retry και orphan reconciliation.

Security και resource limits

Filename και declared content type δεν είναι αξιόπιστη απόδειξη file type. Το identification μπορεί να εξετάσει bytes, αλλά hostile file χρειάζεται application-specific policy, size limits και ίσως malware scanning. Image and video analyzers εκτελούν external tools και μπορούν να καταναλώσουν CPU, RAM και disk. Κράτα queue isolation, timeouts και bounded variants.

Signed blob id αποτρέπει tampering με id και purpose. Δεν κάνει authorization για κάθε viewer. Proxy or redirect controller policy πρέπει να ελέγχει ποιος δικαιούται το application record ή να χρησιμοποιεί intentionally public service.

SOURCE TRACE

Το three-system boundary φαίνεται στα `ActiveStorage::Blob`, `ActiveStorage::Attachment` και `DirectUploadsController`. Ακολούθησε `create_and_upload!`, `create_before_direct_upload!`, attachment commit callbacks και `purge`. Το public setup βρίσκεται στο Active Storage guide.

ΠΑΓΙΔΑ

Μην τρέχεις `purge` μέσα σε database transaction. Κάνει HTTP I/O αφού αλλάξει database state και κρατά το transaction ανοιχτό. Χρησιμοποίησε durable job και reconciler για orphan rows και orphan objects.

ΜΗΧΑΝΙΣΜΟΣ

Active Storage προσφέρει adapters και lifecycle hooks, όχι distributed commit. Η εφαρμογή πρέπει να ορίσει πότε ένα file είναι usable και πώς ανακτά κάθε partial state.

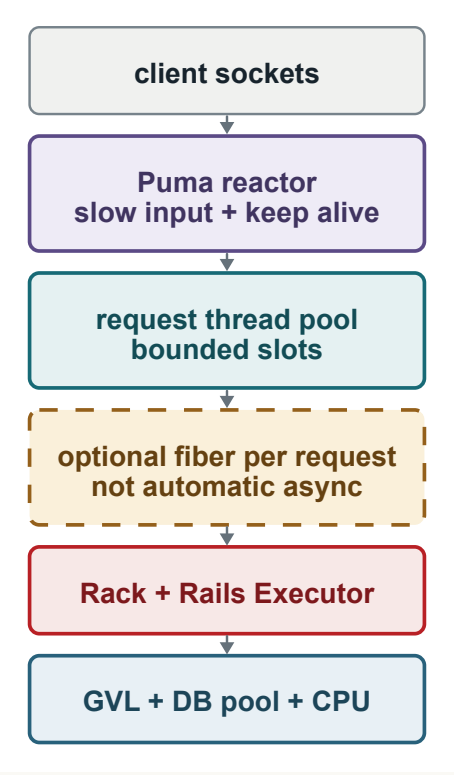
PROBE

Σε isolated service, κόψε την upload σύνδεση μετά το blob create, κλείσε direct upload πριν attach και κάνε το service delete να αποτύχει μετά το blob destroy. Τρέξε cleanup και reconciliation δύο φορές. Assert ότι live attachments δεν χάνονται και ότι κάθε orphan καταλήγει σε observable terminal state.

ΚΕΦΑΛΑΙΟ 27

Puma, GVL, threads και fibers

Η concurrency topology της εφαρμογής είναι processes επί threads επί optional fibers. Κάθε επίπεδο αλλάζει άλλο resource και κανένα setting δεν είναι δωρεάν.



Σχήμα 27.1 · Η Puma process έχει reactor, thread pool και Rails execution ανά request.

Κάθε Puma server process έχει reactor και request thread pool. Ο reactor κρατά clients που δεν έχουν ακόμη δώσει πλήρες request ή περιμένουν keep-alive data, ώστε slow sockets να μη δεσμεύουν application threads χωρίς λόγο. Όταν client είναι έτοιμος, μπαίνει στο thread pool. Processor thread καλεί το Rack app και κρατά το request μέχρι να ολοκληρωθεί το body lifecycle.

Σε cluster mode, master κάνει fork πολλούς workers. Κάθε worker είναι ξεχωριστή Ruby VM με δικό του GVL, heap, Rails objects, thread pool και database pools. Το `workers 4` και `threads 5, 5` δίνουν ως 20 concurrent request slots, όχι ένα pool 5 threads. Προστίθενται reactor, control και library threads.

Τι κάνει και τι δεν κάνει το GVL

Στο MRI, ένα thread ανά process εκτελεί Ruby code κάθε στιγμή υπό Global VM Lock. Blocking network I/O και αρκετές native operations απελευθερώνουν το lock, οπότε άλλο thread προχωρά. Γι' αυτό threads αυξάνουν throughput για Rails requests που περιμένουν database, cache και HTTP. Δύο CPU-heavy Ruby loops όμως δεν τρέχουν πραγματικά παράλληλα στον ίδιο process. Αν αυξήσεις threads, προσθέτεις GVL contention και latency χωρίς νέο CPU execution capacity.

Processes δίνουν parallel Ruby CPU σε πολλούς cores, με τίμημα περισσότερη RAM και ξεχωριστά pools. Η σωστή μίξη εξαρτάται από measured ratio CPU προς wait. Δεν προκύπτει μόνο από core count.

```ruby
started = Process.clock_gettime(Process::CLOCK_MONOTONIC)
cpu_started = Process.clock_gettime(Process::CLOCK_THREAD_CPUTIME_ID)

result = YieldReport.call

wall = Process.clock_gettime(Process::CLOCK_MONOTONIC) - started
cpu = Process.clock_gettime(Process::CLOCK_THREAD_CPUTIME_ID) - cpu_started
Rails.event.notify("seatly.yield_report", wall: wall, cpu: cpu,
  rows: result.length)
```

Μεγάλο wall και μικρό thread CPU υποδεικνύει waiting ή scheduling. Μεγάλα και τα δύο δείχνουν CPU work. Είναι ένδειξη, όχι πλήρες profiler.

Queueing πριν από την εφαρμογή

Όταν όλα τα Puma threads είναι busy, ready clients περιμένουν στο internal todo queue ή στο upstream socket backlog. Το controller instrumentation δεν βλέπει όλο αυτό το pre-request wait. Χρειάζεσαι load balancer timestamps ή Puma stats όπως backlog, running και busy threads. Tail latency μπορεί να αυξάνεται ενώ `process_action` time παραμένει σταθερό.

Αν κάθε request χρειάζεται database, `max_threads` μεγαλύτερο από Active Record pool δημιουργεί δεύτερη ουρά μέσα στο request. Αυτό μπορεί να είναι αποδεκτό για mixed endpoints που δεν κάνουν όλα DB, αλλά πρέπει να είναι συνειδητό. Υπολόγισε και async query, Cable και in-process job threads που αγγίζουν το ίδιο pool.

Fibers δεν σημαίνουν αυτόματα async Rails

Ruby Fiber είναι cooperatively scheduled execution unit μέσα σε thread. Με Fiber Scheduler, supported blocking operations μπορούν να παραχωρούν τον thread χωρίς να μπλοκάρουν όλη την OS thread. Αυτό απαιτεί scheduler και κάθε I/O library να συνεργάζεται. Μία blocking C extension μπορεί ακόμη να δεσμεύσει.

Το Puma 8.0.2 έχει `fiber_per_request`, που τυλίγει κάθε request σε νέο Fiber. Η implementation απλώς δημιουργεί Fiber και κάνει resume γύρω από `handle_request`. Χωρίς scheduler δεν προσθέτει I/O concurrency. Είναι χρήσιμο για καθαρισμό fiber-local state. Το alias `clean_thread_locals` αποκαλύπτει αυτό το intent.

Το Active Support `IsolatedExecutionState` είναι default per thread, αλλά μπορεί να ρυθμιστεί per fiber. Η αλλαγή έχει μεγάλο blast radius: CurrentAttributes, database context και framework caches

πρέπει να ευθυγραμμίζονται με το scheduling model. Μην την κάνεις μόνο επειδή ενεργοποίησες ένα Puma option.

Thread safety ως shared mutation

Controller instances είναι per request, αλλά classes, constants, registries και process caches μοιράζονται. Memoization σε class method με mutable hash, αλλαγή callback chain μετά το boot και reuse non-thread-safe client είναι πραγματικά races. Το GVL δεν δίνει logical atomicity σε read-modify-write sequence και άλλες Ruby implementations έχουν διαφορετικό runtime.

SOURCE TRACE

Το Puma request path βρίσκεται στο tagged `puma_v8.0.2`, ειδικά `lib/puma/server.rb`, `reactor.rb` και `thread_pool.rb`. Το Rails execution state βρίσκεται στο `isolated_execution_state.rb`. Για runtime semantics δες τα επίσημα Ruby 3.4 docs για `Thread` και `Fiber`.

ΠΑΓΙΔΑ

Μην υπολογίζεις capacity ως `workers * threads` και σταματάς εκεί. Πρόσθεσε DB pools, memory per worker, GVL CPU, internal threads, upstream backlog και deadline. Το μικρότερο saturated resource καθορίζει το throughput.

ΜΗΧΑΝΙΣΜΟΣ

Threads κρύβουν I/O wait. Processes δίνουν Ruby CPU parallelism και isolation. Fibers δίνουν cooperative scheduling μόνο όταν υπάρχει scheduler-aware stack. Ρύθμισε το επίπεδο που αντιστοιχεί στο measured bottleneck.

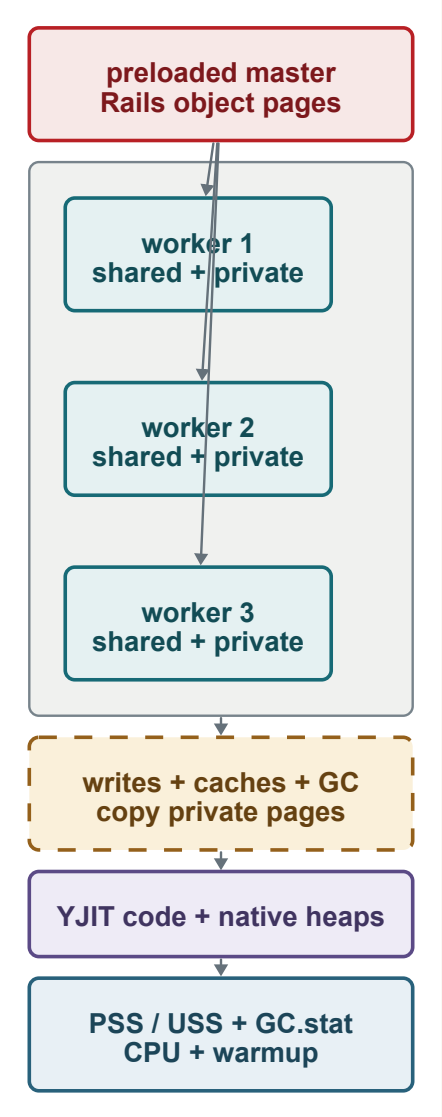
PROBE

Κάνε load test με σταθερό arrival rate και 2, 5, 10 threads ανά worker, κρατώντας σταθερά workers και DB pool. Μέτρησε throughput, p95/p99, Puma backlog, GVL/CPU, checkout wait και RSS. Μετά άλλαξε workers κρατώντας συνολικά slots παρόμοια. Διάλεξε από την καμπύλη, όχι από το μεγαλύτερο throughput σε ένα burst.

ΚΕΦΑΛΑΙΟ 28

Preload, copy-on-write, YJIT και GC

Η μνήμη ενός Rails worker είναι ιστορία allocations, fork sharing, mutations, native heaps και generated machine code. Το RSS μόνο του λέει μικρό μέρος.



Σχήμα 28.1 · Το preloaded master μοιράζεται pages μέχρι workers να τις γράψουν.

Με `preload_app!`, ο Puma master φορτώνει την Rails εφαρμογή πριν κάνει fork. Τα child workers ξεκινούν με virtual pages που το λειτουργικό μπορεί να μοιράζεται. Όταν worker γράψει σε page, το

copy-on-write δημιουργεί private copy. Το preload μειώνει boot duplication όταν πολλά objects παραμένουν αμετάβλητα. Δεν κάνει όλη τη heap μόνιμα shared.

Οτιδήποτε ανοίχτηκε πριν το fork χρειάζεται audit. Database sockets, background threads, telemetry exporters και random state δεν πρέπει να χρησιμοποιούνται αφελώς από parent και children. Τα Puma hooks `before_fork`, `before_worker_boot` και `after_worker_shutdown` ορίζουν process lifecycle. Rails και adapters χειρίζονται αρκετά defaults, αλλά custom clients χρειάζονται explicit disconnect και reconnect.

Τι σπάει το sharing

Class-level caches που γεμίζουν μετά το fork, lazy constant loading, modifying preloaded arrays, per-worker JIT code και ordinary request allocations αυξάνουν private memory. Ακόμη και GC metadata writes μπορούν να λερώσουν pages. Το `freeze` σε μεγάλα read-only structures βοηθά semantics και μερικές φορές COW, αλλά δεν είναι γενικός memory optimizer.

Άθροισμα RSS όλων των workers διπλομετρά shared pages. Σε Linux, PSS μοιράζει το κόστος των shared pages και USS δείχνει private memory. Σε container, κοίτα επίσης cgroup limit και working set. Ένα process μπορεί να έχει σταθερό object count αλλά αυξανόμενο RSS λόγω allocator fragmentation ή native library.

Ruby heap και εξωτερική μνήμη

Η Ruby GC οργανώνει objects σε heap pages και generations. `GC.stat` εκθέτει allocated objects, heap slots, major/minor collections και άλλα counters. Δεν περιλαμβάνει όλη τη native μνήμη που κρατούν database drivers, image libraries ή malloc arenas. `ObjectSpace` δείχνει live Ruby objects, όχι πλήρη resident footprint.

```ruby
snapshot = GC.stat.slice(
  :heap_live_slots,
  :heap_available_slots,
  :total_allocated_objects,
  :major_gc_count,
  :minor_gc_count
)

Rails.event.notify("seatly.gc.snapshot", snapshot)
```

Μην το στέλνεις ανά request. Sample ανά process σε λογικό interval και πρόσθεσε worker identity. Η διαφορά counters έχει μεγαλύτερη αξία από ένα absolute number.

`GC.compact` μπορεί να μετακινήσει movable objects για να μειώσει fragmentation και να βοηθήσει sharing πριν fork σε ορισμένα workloads. Έχει pause cost και δεν μετακινεί κάθε object. Το Puma έχει `nakayoshi_fork` behavior που σχετίζεται με GC/compaction, αλλά πρέπει να benchmark-αριστεί στην πραγματική Ruby έκδοση και heap. Δεν είναι default θεραπεία.

YJIT ως memory-for-speed tradeoff

Στο Rails 8.1 finisher υπάρχει initializer που καλεί `RubyVM::YJIT.enable` όταν `config.yjit` είναι ενεργό και η VM το υποστηρίζει. Η στιγμή είναι μετά από μεγάλο μέρος του boot. YJIT μεταγλωττίζει hot code σε machine code και κρατά code/data memory. Μπορεί να μειώσει CPU και latency μετά το

warmup με πρόσθετη RAM. Cold benchmarks πριν ζεσταθούν methods λένε άλλη ιστορία από long-running workers.

Χρησιμοποίησε `RubyVM::YJIT.runtime_stats` μόνο όταν είναι διαθέσιμο και κράτα version-aware keys. Το stats schema δεν είναι application contract. Μέτρα request CPU, latency, warmup time, code memory και total PSS με YJIT on/off.

Leak, retention, fragmentation ή cache

True leak σημαίνει objects που παραμένουν reachable χωρίς λόγο. Retention μπορεί να είναι bounded cache που δεν έφτασε plateau. Fragmentation σημαίνει free space που allocator δεν επιστρέφει. Native leak δεν εμφανίζεται στο Ruby object graph. JIT growth είναι αναμενόμενο μέχρι plateau. Η θεραπεία διαφέρει.

Πάρε time series: RSS/PSS, live slots, old objects, malloc bytes όπου διαθέσιμο, request count και cache sizes. Heap dump δύο φορές με sensitive-data process και compare retained classes. Μην ανεβάζεις production heap dump σε εξωτερικό service χωρίς privacy approval. Περιέχει strings και user data.

Worker recycling

Periodic restart περιορίζει unbounded growth και δίνει operational safety, αλλά κρύβει root cause και δημιουργεί warmup cost. Χρησιμοποίησέ το ως guardrail με jitter, drain και alert, ενώ βρίσκεις retention. Αν όλοι οι workers recycle μαζί, χάνεις capacity και COW benefits ταυτόχρονα.

SOURCE TRACE

Το process lifecycle και preload contract βρίσκονται στο `Puma v8.0.2 README και source`. Το Rails YJIT initializer είναι στο `application/finisher.rb`. Για counters και compaction χρησιμοποίησε τα επίσημα `Ruby 3.4 GC docs` και το API της ακριβούς Ruby VM.

ΠΑΓΙΔΑ

Μην συγκρίνεις συνολικό RSS preloaded και non-preloaded clusters χωρίς shared memory metric. Μπορεί να συμπεράνεις το αντίθετο από την πραγματική physical memory κατανάλωση.

ΜΗΧΑΝΙΣΜΟΣ

Memory tuning χρειάζεται attribution. Ruby live objects, shared pages, private dirty pages, native allocations και YJIT code είναι διαφορετικοί λογαριασμοί.

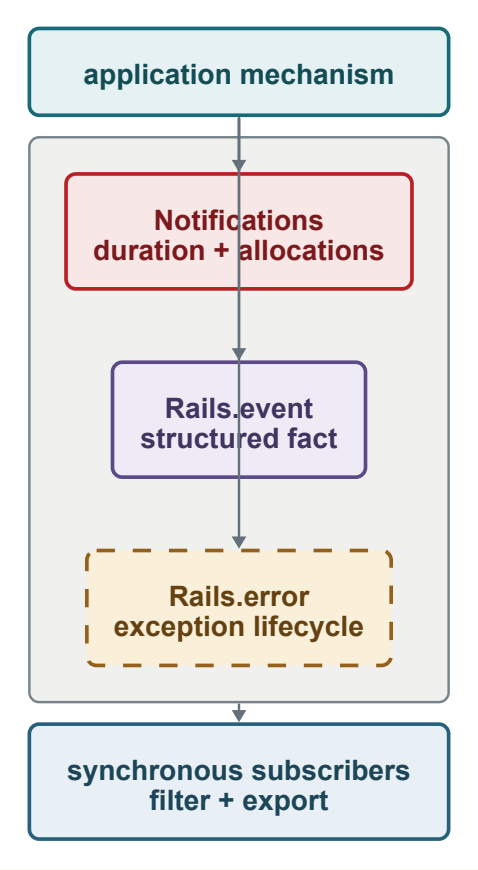
PROBE

Boot-αρε δύο identical staging processes με και χωρίς preload/YJIT. Μετρήσε master και workers σε cold start, μετά 1.000 και 50.000 representative requests. Κατέγραψε PSS/USS όπου υποστηρίζεται, `GC.stat`, YJIT stats, CPU και p99. Κάνε fork hooks fail fast για custom clients που κληρονόμησαν ανοιχτό socket.

ΚΕΦΑΛΑΙΟ 29

Notifications, Rails.event και Rails.error

Duration spans, structured facts και exceptions είναι τρία διαφορετικά σήματα. Το Rails έχει διαφορετικό μηχανισμό για το καθένα.



Σχήμα 29.1 · Τα τρία signal paths και τα διαφορετικά subscriber contracts.

Το `ActiveSupport::Notifications` μετρά block lifecycle. Το `Rails.event` του Rails 8.1 εκπέμπει structured point event. Το `Rails.error` αναφέρει exception με handled status, severity, source και execution context. Αν τα μετατρέψεις όλα σε log line, χάνεις τις διαφορές που κάνουν χρήσιμο το καθένα.

Notifications: synchronous span bus

Η `instrument(name, payload)` ελέγχει αν υπάρχει listener. Αν όχι, κάνει μόνο yield και αποφεύγει μεγάλο μέρος του overhead. Αν υπάρχει, ο Instrumenter χτίζει handle, κάνει start, τρέχει το block και

finish σε ensure. Σε exception προσθέτει class/message και exception object στο payload πριν ξανασηκώσει.

Το Event μετρά monotonic duration, thread CPU time, allocations και GC time όπου υποστηρίζονται. Οι subscribers τρέχουν στο publishing path. Αργός subscriber αυξάνει το latency της operation. Exporter που κάνει synchronous network call μέσα σε `sql.active_record` μπορεί να γίνει χειρότερος από το query.

```ruby
subscriber = ActiveSupport::Notifications.monotonic_subscribe(
  "seatly.confirm"
) do |_name, started, finished, _id, payload|
  Metrics.observe(:confirm_seconds, finished - started,
    outcome: payload.fetch(:outcome))
end
```

Στο production ο subscriber ζει συνήθως όλη τη process. Temporary subscribe και unsubscribe ακυρώνει internal fanout caches και δεν είναι hot-path pattern.

Rails.event: structured fact

`Rails.event.notify` φτιάχνει event hash με name, payload, tags, context, realtime nanosecond timestamp και source location. Hash payload keys κανονικοποιούνται και περνούν από configured parameter filtering. Event objects περνούν ως έχουν και ο subscriber αναλαμβάνει serialization και sensitive filtering.

Tags είναι nested fiber-local stack για domain categories. Context προορίζεται για request ή job metadata και καθαρίζεται στα framework boundaries. Subscribers έχουν optional filter proc και `emit(event)` contract. Αν subscriber αποτύχει, ο reporter συνήθως αναφέρει το subscriber error μέσω error reporter αντί να σπάσει application flow. Σε debug mode ή configured raise behavior το contract διαφέρει.

```ruby
Rails.event.tagged(component: "reservations") do
  Rails.event.notify(
    "reservation.confirmed",
    reservation_id: reservation.id,
    confirmation_version: reservation.lock_version
  )
end
```

Αυτό είναι operational event, όχι durable domain event. Αν process πέσει πριν ο subscriber export, δεν υπάρχει replay guarantee. Για business integration χρησιμοποίησε outbox.

Rails.error: exception lifecycle

`Rails.error.handle` καταπίνει matching exception και επιστρέφει fallback. `record` αναφέρει και ξανασηκώνει. `report` δέχεται exception instance και δηλώνει αν είναι handled. `unexpected` κάνει report σε production-like mode, αλλά σε debug mode σηκώνει wrapper ώστε η παραβίαση assumption να φανεί.

Το reporter συγχωνεύει execution context, περνά context middleware και καλεί subscribers. Μαρκάρει exception και cause chain ώστε ίδιο object να μη σταλεί ξανά από πολλά framework layers. Αυτό σημαίνει ότι δύο διαφορετικά exception instances με ίδιο message είναι δύο reports, ενώ re-report του ίδιου object αγνοείται.

Cardinality και causality

Request id είναι καλό context για trace correlation, κακό metric label. User id μπορεί να είναι sensitive και high-cardinality. Exception message μπορεί να περιέχει input. Σχεδίασε bounded dimensions: operation, outcome, exception class, queue και adapter. Κράτα unique ids σε searchable event body με retention policy.

Ένα useful causal chain ενώνει request id, domain intent id, outbox event id, job id και provider idempotency key. Δεν χρειάζεται να είναι όλα metric labels. Χρειάζεται να υπάρχουν σε structured records που επιτρέπουν trace χωρίς να καταγράφουν secrets.

Subscriber cost και failure policy

Οι Notifications subscribers τρέχουν στο instrumented path. Αν κάνουν blocking network I/O, το latency ανήκει στο request ή job που εξέπεμψε το signal. Η ασύγχρονη εξαγωγή θέλει bounded buffer, drop policy και counter για dropped events. Χωρίς αυτά, telemetry outage μπορεί να γίνει application outage ή memory leak. Μέτρησε το instrumentation με ενεργούς production-like subscribers, όχι μόνο το κόστος ενός κενού block.

Το Rails.event έχει δικό του reporter και context stack, αλλά παραμένει structured observation mechanism. Ένα subscriber failure πρέπει να έχει ξεκάθαρη πολιτική: propagates, αναφέρεται και συνεχίζει ή απομονώνεται από adapter. Επιβεβαίωσέ την στην ακριβή Rails έκδοση. Μην αφήνεις την επιχειρησιακή ορθότητα να εξαρτάται από το αν exporter ήταν διαθέσιμος.

Context cleanup ως invariant

Request και job context πρέπει να καθαρίζονται ακόμη σε exception. Ένα leaked tag από προηγούμενο tenant είναι και observability corruption και πιθανό privacy incident. Γράψε test που χρησιμοποιεί το ίδιο worker thread για δύο executions, ρίχνει exception στο πρώτο και επιβεβαιώνει ότι το δεύτερο ξεκινά με καθαρό context. Αυτό δοκιμάζει το execution wrapper boundary που είδαμε στην αρχή του βιβλίου, κλείνοντας συνειδητά τον κύκλο από runtime σε telemetry.

Nested spans και clock semantics

Τα Notifications events σχηματίζουν φυσικά nested durations όταν instrumented blocks καλούν άλλα instrumented blocks. Αυτό δεν σημαίνει ότι το άθροισμα των child durations ισούται με τον parent. Υπάρχει δικό του work, overlap από async operations και subscriber overhead. Για flame-like ανάλυση χρειάζεσαι explicit parent identity ή trace adapter, όχι απλή αφαίρεση timestamps από ανεξάρτητα events.

Χρησιμοποίησε monotonic duration για elapsed time. Wall-clock timestamp είναι χρήσιμο για συσχέτιση με logs, αλλά μπορεί να αλλάξει λόγω clock adjustment. CPU time και allocation count απαντούν άλλες ερωτήσεις. Ένα span με 800 ms wall και 5 ms CPU πιθανότατα περίμενε. Ένα με wall

κοντά στο CPU κατανάλωσε Ruby ή native compute. Αυτή είναι ένδειξη για το επόμενο probe, όχι αυτόματη root cause.

Schema για operational events

Ένα structured event name χρειάζεται owner και field schema όπως κάθε άλλο interface. Πρόσθεσε fields με backward-compatible τρόπο και κράτα bounded types. Αν dashboard περιμένει `duration_ms` και νέο release στείλει string ή αλλάξει unit, το observability χαλά ακριβώς μέσα στο incident όπου το χρειάζεσαι. Βάλε contract test στον exporter και sample πραγματικό serialized event, χωρίς sensitive values, στο release packet.

SOURCE TRACE

Τα span internals βρίσκονται στα `notifications.rb` και `notifications/instrumenter.rb`. Το structured API είναι το `event_reporter.rb`. Το exception contract βρίσκεται στο `error_reporter.rb`.

ΠΑΓΙΔΑ

Κάθε subscriber είναι application code στο signal path. Κάνε serialization bounded, buffering explicit και failures observable. Μην αφήσεις telemetry outage να γεμίσει unbounded in-memory queue.

ΜΗΧΑΝΙΣΜΟΣ

Χρησιμοποίησε Notifications για διάρκεια, Rails.event για structured fact και Rails.error για exception lifecycle. Durable business messaging ανήκει αλλού.

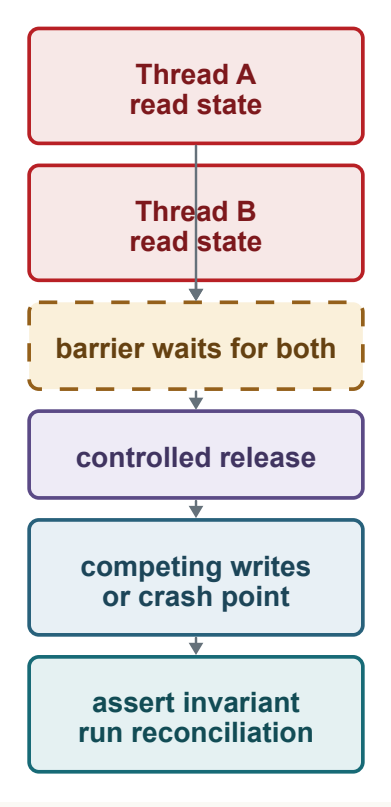
PROBE

Φτιάξε test subscriber για κάθε μηχανισμό. Σήκωσε exception μέσα σε instrumented block, κάνε event subscriber να αποτύχει και ανέφερε ίδιο exception δύο φορές. Κατέγραψε τι propagates, τι γίνεται report και ποιο context καθαρίζεται στο επόμενο executor cycle. Μέτρησε επίσης subscriber overhead με και χωρίς listener.

ΚΕΦΑΛΑΙΟ 30

Deterministic races και fault injection

To concurrent test δεν πρέπει να ελπίζει ότι ο scheduler θα πετύχει το race. Πρέπει να κατασκευάζει τη σειρά γεγονότων με barriers.



Σχήμα 30.1 · Τα barriers τοποθετούν δύο executions ακριβώς μέσα στο failure window.

Ένα `sleep 0.1` δεν αποδεικνύει ordering. Σε γρήγορο laptop μπορεί να είναι αρκετό, σε CI όχι. Η Ruby `Queue` και δύο signals επιτρέπουν να σταματήσεις κάθε thread μετά τη read και να τα απελευθερώσεις όταν και τα δύο βρίσκονται στο ίδιο logical σημείο. Το test γίνεται repeatable και αποτυγχάνει για τη σωστή αιτία.

```ruby
ready = Queue.new
release = Queue.new

workers = 2.times.map do
  Thread.new do
    Rails.application.executor.wrap do
      ActiveRecord::Base.connection_pool.with_connection do
        reservation = Reservation.find(42)
        ready << reservation.lock_version
        release.pop
        reservation.update!(status: :confirmed)
      end
    end
  rescue ActiveRecord::StaleObjectError
    :conflict
  end
end

2.times { ready.pop }
2.times { release << true }
outcomes = workers.map(&:value)
puts outcomes.inspect
```

Το test χρειάζεται pool τουλάχιστον δύο connections και committed fixture state ορατό και στα δύο. Transactional test που κρατά data uncommitted σε άλλο connection μπορεί να κρύψει rows ή να serialization-άρει την εκτέλεση. Για αυτό το μικρό scope απενεργοποίησε transactional tests και καθάρισε data με explicit strategy, χωρίς να αλλάξεις global suite τυχαία.

Schedule το interleaving, όχι το αποτέλεσμα

Κατασκεύασε barriers στα semantic σημεία: after read, after external effect, before commit, after claim, before acknowledgement. Μην βάζεις test hooks σε private Rails classes αν μπορείς να βάλεις interface στη δική σου operation. Ένα injected collaborator με `after_charge` hook κάνει το crash window ελεγχόμενο χωρίς monkey patch στον adapter.

```ruby
class PaymentProbe
  def initialize(after_charge: nil)
    @after_charge = after_charge
  end

  def charge(key:)
    result = FakeGateway.charge(key: key)
    @after_charge&.call(result)
    result
  end
end
```

Στο test, το hook μπορεί να σηκώσει process-like exception μετά το provider effect. Σε production δεν περνάς hook. Το seam ανήκει στην application boundary και ελέγχει business ambiguity.

Crash δεν είναι πάντα StandardError

Ένα `raise` τρέχει ensure blocks. Hard process kill, OOM ή host loss δεν τα τρέχει. Για να ελέγξεις durable recovery, χρειάζεσαι child process ή ξεχωριστό worker που τερματίζεται χωρίς graceful

cleanup. Κάν' το μόνο σε isolated test resources. Το unit fault injection ελέγχει control flow. Το process test ελέγχει τι πραγματικά έμεινε durable.

Network timeout έχει τρία διαφορετικά σημεία: πριν ο server λάβει request, ενώ το επεξεργάζεται και αφού ολοκλήρωσε αλλά πριν φτάσει response. Fake που πάντα σηκώνει πριν το effect ελέγχει μόνο την εύκολη περίπτωση. Το ambiguous-after-effect case είναι εκεί που χρειάζεται idempotency ή status lookup.

Model-based state tests

Για outbox, saga ή queue consumer, όρισε μικρό pure reference model με legal states και commands. Παρήγαγε sequences από deliver, duplicate, timeout, retry, late success και compensation. Μετά από κάθε βήμα σύγκρινε database state με model invariants. Δεν χρειάζεται νέο property-testing dependency για να αρχίσεις. Ένας deterministic random seed και εκατό sequences είναι ήδη καλύτερα από πέντε happy paths.

Time και deadlines

Χρησιμοποίησε Rails time helpers για expiration logic σε single-threaded tests. Σε concurrent tests, global time travel μπορεί να επηρεάσει όλα τα threads. Προτίμησε injected clock στην application state machine. Η database `NOW()` και Ruby `Time.current` μπορούν επίσης να διαφέρουν. Διάλεξε ποιος clock κατέχει lease expiry και έλεγξέ τον στο integration boundary.

Assert observables

Μην κάνεις assert ότι κλήθηκε private `commit_records`. Assert ότι μετά από rollback δεν υπάρχει outbox row, μετά από duplicate delivery υπάρχει ένα effect και μετά από deadlock retry ισχύει invariant. Source-level tests έχουν νόημα μόνο σε framework adapter που συνειδητά συντηρείς.

Hard crash matrix

Exception injection δοκιμάζει ensure και rescue. Δεν δοκιμάζει process loss. Για durable workflow χρειάζεσαι ξεχωριστό process που μπορείς να τερματίσεις σε named point αφού έχει γράψει durable marker ότι έφτασε εκεί. Ο parent test περιμένει το marker με deadline, στέλνει termination και ξεκινά νέο worker ή reconciler. Μετά ελέγχει μόνο database, queue και external fake state.

Το fake external system πρέπει να μοντελοποιεί ambiguous success. Να μπορεί να καταγράψει effect και μετά να κόψει τη σύνδεση πριν επιστρέψει response. Αν το fake απλώς σηκώνει πριν από κάθε effect, δεν δοκιμάζεις το δύσκολο window. Πρόσθεσε stable idempotency key, lookup endpoint και delayed response ώστε να ελέγξεις retry, query-before-repeat και late callback.

Barriers με δικό τους deadline

Κάθε barrier χρειάζεται state names που εμφανίζονται σε failure output: `a_read`, `b_read`, `release_writes`, `a_committed`. Ένα raw pair από queues χωρίς diagnostics κάνει το hung CI ακατανόητο. Βάλε monotonic deadline σε κάθε αναμονή και, στην αποτυχία, τύπωσε μόνο thread status, connection pool stats και barrier state. Μην κάνεις blind thread dump σε suite που μπορεί να περιέχει sensitive request data.

Τρέξε το deterministic test με πραγματικό adapter. SQLite in-memory και PostgreSQL δεν έχουν ίδιο locking model. Fast unit model tests καλύπτουν πολλές sequences. Λίγα adapter integration cases αποδεικνύουν τα κρίσιμα windows.

SOURCE TRACE

Τα Rails test primitives βρίσκονται στο `activesupport/lib/active_support/testing` και τα Active Record transaction test helpers στο `activerecord/lib/active_record/test_fixtures.rb`. Για τον μηχανισμό που δοκιμάζεις, ακολούθησε τα chapters 11, 18 και 22. Τα barriers εδώ είναι Ruby standard `Queue`, όχι Rails internal.

ΠΑΓΙΔΑ

Ένα test που περνά 10.000 φορές χωρίς να αποδείξεις ότι μπήκε στο intended window μπορεί να μην δοκιμάζει τίποτα. Κατέγραψε barrier states και βάλε timeout ώστε broken synchronization να αποτυγχάνει αντί να κρεμά το suite.

ΜΗΧΑΝΙΣΜΟΣ

Το deterministic concurrency test ελέγχει μία συγκεκριμένη partial order. Το fault-injection matrix καλύπτει πολλά crash points. Μαζί μετατρέπουν ένα vague «είναι idempotent» σε εκτελέσιμη απόδειξη.

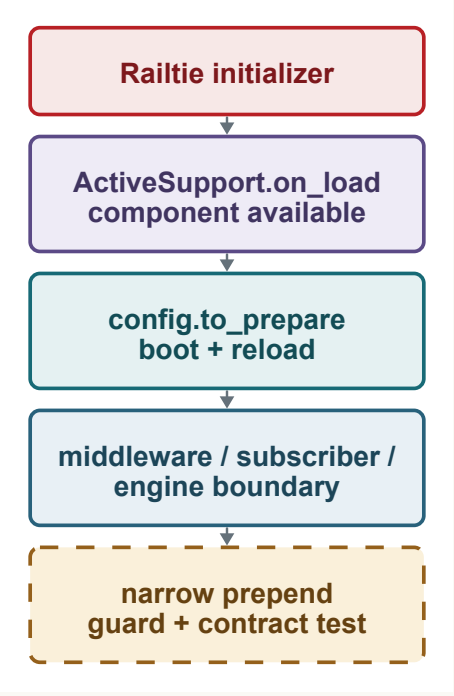
PROBE

Διάλεξε ένα critical job και σχεδίασε πέντε named fault points. Πρόσθεσε barrier ή injectable hook σε application-owned seam. Τρέξε κάθε point δύο φορές και μετά τον reconciler. Assert final invariant, duplicate tolerance, terminal state και bounded retry count. Βάλε timeout σε κάθε thread join.

ΚΕΦΑΛΑΙΟ 31

Railties, engines, load hooks και source debugging

Η ασφαλής επέκταση του Rails χρησιμοποιεί public lifecycle hooks και στενούς adapters. Το monkey patch είναι τελευταία επιλογή με δικό του compatibility test.



Σχήμα 31.1 · Τα public extension points γύρω από framework load και application reload.

Ένα `Rails::Railtie` μπορεί να προσθέσει configuration, initializers, rake tasks, generators και console hooks χωρίς να είναι full application. Ένα `Rails::Engine` είναι railtie με routes, controllers, models, views, assets και paths που συμμετέχουν στην host application. Το `isolate_namespace` περιορίζει naming και routing collisions, αλλά δεν δημιουργεί process ή database isolation.

Για cross-cutting gem που προσθέτει subscriber, railtie αρκεί. Για mountable sub-application με routes και UI, engine έχει νόημα. Μην κάνεις engine ένα folder μόνο για να φαίνεται modular. Η boundary πρέπει να έχει πραγματική ownership και public surface.

Active Support load hooks

Το `ActiveSupport.on_load(:active_record)` καταχωρίζει block που θα τρέξει όταν το framework component καλέσει `run_load_hooks`. Αν το component έχει ήδη φορτωθεί, ο hook τρέχει και για τα ήδη loaded bases. Αυτό αποφεύγει να κάνεις eager `require "active_record"` μόνο για να το επεκτείνεις.

```ruby
ActiveSupport.on_load(:active_record) do
  include Seatly::RecordTracing
end
```

Το block συχνά αξιολογείται στο context της loaded base. Έλεγξε το συγκεκριμένο hook contract. Κράτα την επέκταση idempotent, επειδή reload ή multiple load bases μπορούν να την ενεργοποιήσουν παραπάνω από μία φορά.

Το `config.to_prepare` τρέχει στο boot και ξανά πριν από application execution όταν γίνεται reload. Είναι σωστό για wiring που αναφέρεται σε reloadable classes. Το block πρέπει να μπορεί να ξανατρέξει χωρίς να διπλασιάσει subscribers, callbacks ή routes. Αν γράψεις `subscribe` κάθε φορά χωρίς unsubscribe ή stable registry, κάθε reload προσθέτει observer.

Narrow prepend με guard

Όταν δεν υπάρχει public hook και χρειάζεται framework patch, βάλε το σε ένα module, σε exact target, μετά το relevant load hook. Έλεγξε owner, parameters και version. Το `prepend` διατηρεί `super` chain καλύτερα από alias-method tricks, αλλά παραμένει coupling σε implementation.

```ruby
module SeatlySelectTracing
  def select_all(*args, **kwargs)
    Rails.event.tagged(adapter: adapter_name) { super }
  end
end

ActiveSupport.on_load(:active_record) do
  target = ActiveRecord::ConnectionAdapters::AbstractAdapter
  target.prepend(SeatlySelectTracing) unless target < SeatlySelectTracing
end
```

Το παράδειγμα δείχνει guard, αλλά το target method μπορεί να αλλάξει signature ή να μην είναι η σωστή hot path σε μελλοντική έκδοση. Ένα Notifications subscriber στο `sql.active_record` είναι συνήθως σταθερότερη λύση. Patch μόνο όταν το public signal δεν μπορεί να δώσει την απαραίτητη εγγύηση.

Source debugging στο runtime

Ξεκίνα με `method.owner`, `source_location`, `parameters` και `super_method`. Για generated methods, owner μπορεί να είναι anonymous module και location να δείχνει generated evaluation. Με `Module#ancestors` βρίσκεις inclusion order. Με `$LOADED_FEATURES.grep` βλέπεις ποιο file φορτώθηκε. Με Zeitwerk logger μπορείς να δεις constant paths σε development diagnostic session.

Το `TracePoint` μπορεί να παρακολουθήσει calls, returns και exceptions, αλλά global trace έχει μεγάλο overhead. Φίλτραρε path, class και thread και ενεργοποίησέ το μόνο γύρω από μικρό reproduction. Μην το αφήσεις μόνιμα σε production request traffic χωρίς measured budget.

```ruby
target = Reservation.instance_method(:save!)
while target
  puts [target.owner, target.source_location, target.parameters].inspect
  target = target.super_method
end
```

Compatibility adapter

Απομόνωσε private integration σε ένα file με:

1. supported Rails version range
2. boot-time signature assertion
3. behavior contract tests
4. comment με tagged source path και reason
5. kill switch ή fallback

Μην σκορπίσεις `respond_to?` checks σε business code. Ένας adapter απορροφά version branches και αφαιρείται όταν το upstream public API καλύψει την ανάγκη.

Engine boundaries στο reload

Engine code μπορεί να είναι eager-loaded ή reloadable ανά path configuration. Objects αποθηκευμένα σε initializers, registries ή host class attributes μπορούν να δείχνουν παλιές classes μετά unload. Αποθήκευσε stable names ή rebuild wiring στο prepare hook. Μην cache-άρεις subclasses με custom global array όταν το `DescendantsTracker` και reload lifecycle έχουν δικές τους rules.

Idempotent wiring, όχι μόνο idempotent code

Ένα callback body μπορεί να είναι ασφαλές να τρέξει δύο φορές, ενώ η εγγραφή του callback δύο φορές να μην είναι. Σε κάθε reload μέτρα subscribers, middleware entries και callback filters. Αν registration API επιστρέφει handle, φύλαξέ το σε non-reloadable owner και κάνε unsubscribe πριν από νέο subscribe. Αν δεν υπάρχει handle, βάλε stable named object ή guard που ελέγχει την πραγματική registry, όχι boolean μέσα σε reloadable class.

Το `to_prepare` μπορεί να τρέξει στο boot και ξανά πριν από application execution όταν έγινε reload. Το block πρέπει να δουλεύει και στις δύο φάσεις. Μην υποθέτεις ότι υπάρχει request, current tenant ή checked-out connection. Πέρασε μόνο stable configuration και επίλυσε reloadable constants μέσα στο block κάθε φορά.

Patch surface budget

Κάθε private prepend έχει τρία κόστη: πόσα methods αγγίζει, πόσο συχνά αλλάζει ο upstream owner και πόσο δύσκολα παρατηρείται λάθος behavior. Βάλε budget. Ένα override ενός μικρού leaf method με exact parameters και kill switch είναι διαχειρίσιμο. Patch σε κεντρικό dispatcher με πολλούς adapters χρειάζεται πολύ ισχυρότερο contract suite ή upstream contribution.

Στο upgrade, σύγκρινε owner, parameters, source location και ένα μικρό source digest γύρω από το target. Το digest είναι alarm, όχι compatibility proof. Μετά τρέξε behavior tests και το failure path με το patch απενεργοποιημένο. Σκοπός δεν είναι να δέσεις την εφαρμογή σε byte-identical source. Είναι να μη γίνει μια ιδιωτική αλλαγή σιωπηλό production semantic change.

SOURCE TRACE

Τα extension primitives βρίσκονται στα `rails/railtie.rb`, `rails/engine.rb` και `active_support/lazy_load_hooks.rb`. Το prepare lifecycle συνδέεται στο `Rails::Application::Finisher` και στον Reloader. Διάβασε πάντα tagged source της έκδοσης που τρέχεις.

ΠΑΓΙΔΑ

Μην κάνεις source debugging με broad logging sensitive args ή SQL binds. Η μεγαλύτερη ορατότητα αυξάνει και το privacy surface. Φίλτραρε πριν από export.

ΜΗΧΑΝΙΣΜΟΣ

Public hooks δίνουν lifecycle contract. Private prepend δίνει temporary implementation coupling. Αν δεν μπορείς να γράψεις compatibility test και fallback, δεν είσαι έτοιμος να κρατήσεις το patch.

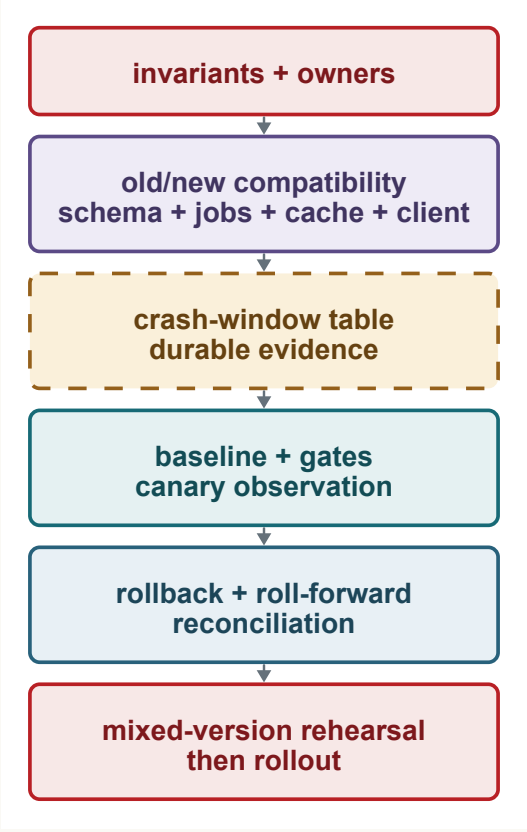
PROBE

Για κάθε local Rails patch, τύπωσε target owner, parameters και source location στο boot και κάνε fail fast σε mismatch. Τρέξε suite μία φορά με patch και μία με kill switch. Σε development κάνε δύο reloads και μέτρησε subscribers ή callbacks ώστε να αποδείξεις ότι το wiring παραμένει μονό.

ΚΕΦΑΛΑΙΟ 32

To correctness packet ενός rolling release

Ένα δύσκολο release δεν εγκρίνεται επειδή πέρασαν tests. Εγκρίνεται όταν έχει μικρό, ελέγξιμο πακέτο αποδείξεων για overlap, failures, observation και rollback.



Σχήμα 32.1 · To correctness packet ενώνει compatibility matrix, fault table και release gates.

Το Seatly θέλει να αντικαταστήσει το παλιό boolean `ticket_issued` με versioned ticket workflow, να μεταφέρει την έκδοση σε continuation job και να δείχνει realtime status. Για ώρες θα συνυπάρχουν old web, new web, old jobs, new jobs, παλιές cache entries και browsers που άνοιξαν πριν το deploy. Η migration που περνά μόνη της σε test δεν απαντά αν αυτό το overlap είναι ασφαλές.

Το correctness packet είναι repository artifact με έξι sections. Δεν χρειάζεται βαρύ process. Χρειάζεται ακριβείς claims που μπορείς να διαψεύσεις.

1. Invariants και owners

Γράψε πρώτα τι δεν επιτρέπεται να σπάσει:

- μία final έκδοση εισιτηρίου ανά reservation και confirmation version
- confirmed reservation δεν χάνει ποτέ durable ticket intent
- duplicate jobs δεν παράγουν δεύτερο provider effect
- old client μπορεί να διαβάσει status κατά το overlap
- rollback code μπορεί να διαβάσει rows που έγραψε το new release

Δίπλα σε κάθε invariant γράψε owner. Unique index, outbox row, provider idempotency key, tolerant serializer ή API representation. Αν owner είναι «ο controller το προσέχει», το invariant δεν έχει ακόμη production boundary.

2. Compatibility matrix

Η matrix έχει producers στις γραμμές και consumers στις στήλες. Δεν αρκεί old app versus new schema.

Producer	Old consumer	New consumer	Απόδειξη
old web row	διαβάζει	διαβάζει	dual-read test
new web row	διαβάζει fallback	διαβάζει version	rollback fixture
old job payload	τρέχει	τρέχει	serialized fixture
new job payload	αγνοεί new optional field	τρέχει	mixed-worker test
old cache entry	διαβάζει	miss ή tolerant decode	namespace test
new Turbo message	safe ignore	version apply	browser contract test

Για delayed jobs, matrix horizon είναι oldest scheduled time συν retry window. Για browser protocol, είναι session/cache lifetime ή explicit min supported client. Για schema, είναι όλο το rolling και rollback window.

3. Expand, write, backfill, enforce, contract

Πρώτο release προσθέτει nullable `ticket_version` και `ticket_state` χωρίς να αφαιρέσει boolean. New code dual-reads: προτιμά versioned state, αλλιώς μεταφράζει boolean. Στη write path γράφει και τα δύο representations. Μετά μετρά divergence.

Backfill δουλεύει σε bounded batches με cursor και conditional update ώστε να μην αντικαταστήσει νεότερο concurrent write. Continuation cursor αποθηκεύει next id, αλλά η operation είναι idempotent αν row ξαναεμφανιστεί.

```ruby
scope = Reservation.where(ticket_state: nil).where(id: cursor...)

scope.order(:id).limit(500).each do |reservation|
  state = reservation.ticket_issued? ? "issued" : "pending"
  Reservation.where(id: reservation.id, ticket_state: nil)
    .update_all(ticket_state: state, ticket_version: 1)
end
```

Μετά το observed zero backlog, νέο constraint μπορεί να validated ή enforced με adapter-safe staged method. Contract removal γίνεται σε ξεχωριστό release αφού rollback window κλείσει. Old column, old serializer branch και old cache namespace δεν αφαιρούνται μαζί με το πρώτο success graph.

4. Failure table

Για κάθε transition γράψε before, ambiguous point και recovery:

Transition	Crash point	Durable evidence	Recovery
confirm -> intent	πριν commit	τίποτα	retry request
intent -> job	μετά commit	outbox event	dispatcher
job -> provider	μετά provider success	command id	status query/dedupe
issued -> broadcast	πριν transmit	canonical row	refresh snapshot
backfill row	πριν cursor advance	conditional row state	replay batch

Αυτή η table γίνεται τα fault-injection tests του κεφαλαίου 30. Αν ένα row δεν έχει durable evidence ή recovery owner, το release έχει άγνωστο terminal state.

5. Observability και gates

Πριν το rollout όρισε bounded signals και abort thresholds. Schema/cache errors, job deserialization failures, dual-write divergence, outbox oldest age, provider unknown outcomes, continuation resumptions, pool wait και realtime gap refreshes. Baseline τα metrics πριν την αλλαγή. Ένα absolute threshold χωρίς baseline μπορεί να είναι αυθαίρετο.

Canary πρέπει να περιλαμβάνει worker, όχι μόνο web. Αν new web παράγει payload που καταναλώνουν αποκλειστικά old workers, το dangerous edge δεν δοκιμάστηκε. Το canary data πρέπει να περνά πραγματικό writer, queue, cache και client path χωρίς να προκαλεί πραγματική χρέωση.

6. Rollback και roll-forward

Rollback απαντά ποιο binary επανέρχεται, ποια schema μένει additive και πώς old code διαβάζει new rows. Δεν περιλαμβάνει άμεσο drop column ή αντιστροφή μεγάλου backfill. Roll-forward απαντά πώς διορθώνεται bad derived data με νέο version. Για external effects, rollback code δεν μπορεί να ξεχρεώσει τον provider. Χρειάζεται compensation state.

Κάνε rehearsal με old και new processes ταυτόχρονα. Serialize old payload και κατανάλωσέ το στο new. Γράψε new row και διάβασέ το στο old. Ζέστανε old prepared statements, τρέξε additive migration και επανάλαβε. Κράτα παλιό browser connected ενώ βγαίνουν new broadcasts. Αυτό είναι compatibility testing, όχι απλώς smoke.

Η τελική απόδειξη

Το packet δεν υπόσχεται ότι δεν θα συμβεί άγνωστο failure. Περιορίζει το known state space και δίνει γρήγορη απόφαση. Για κάθε boundary ξέρουμε mechanism, observable signal και recovery. Αυτή είναι η συνέχεια της τελευταίας πρότασης του πρώτου τόμου: βρίσκουμε ποιος κρατά την εγγύηση και τώρα μπορούμε να δείξουμε πώς την κρατά, πώς σπάει και πώς το αποδεικνύουμε πριν το production.

SOURCE TRACE

Τα framework edges του packet είναι το migration API στο `connection_adapters/abstract/schema_statements.rb`, το schema cache στο `schema_cache.rb`, το job envelope στο `active_job/core.rb` και το initialization/reload lifecycle στο `Rails::Application::Finisher`. Το overall upgrade contract περιγράφεται στο `Upgrading Ruby on Rails`. Adapter-specific online DDL πρέπει να ελεγχθεί στην επίσημη database τεκμηρίωση.

ΠΑΓΙΔΑ

Μην κάνεις destructive contract migration επειδή το dashboard δείχνει zero για λίγα λεπτά. Χρειάζεσαι πλήρες maximum age window, successful rollback rehearsal και επιβεβαίωση ότι κανένας delayed producer δεν γράφει πια το old contract.

ΜΗΧΑΝΙΣΜΟΣ

Το correctness packet συνδέει invariant, compatibility, fault, signal και recovery σε μία ελέγξιμη αλυσίδα. Αν ένα claim δεν έχει proof ή owner, δεν είναι release guarantee.

PROBE

Πάρε την επόμενη schema ή job-contract αλλαγή του Seatly ή της εφαρμογής σου και γράψε το εξασέλιδο packet πριν τον κώδικα. Μετέτρεψε κάθε failure-table row σε test. Τρέξε mixed old/new rehearsal και rollback read test. Μόνο μετά όρισε canary percentage και production gates.