Pragmatic DDD Guide

Oct 31, 2026 · @kodebot

This guide says how we design the inside of a microservice. We use a light form of Domain-Driven Design (DDD). DDD means two things here. Write the code in the business's own words. Put each business rule inside the object it is about.

The guide has two aims. Every business rule lives in exactly one place. A design mistake can be fixed inside one feature of one microservice, without changing the others.

Our stack is Spring Boot 4 (Spring Framework 7, Spring Data MongoDB 5, Spring Security 7) on Amazon DocumentDB. Every example uses it, and an italic line under each names the Spring feature doing the work. Besides Spring, only our test tools are named: Testcontainers, WireMock and Floci. Every rule has the same shape: the situation, the code most people write first, what goes wrong, the code we want, and why. Read the walkthrough and the rules in order once. After that, the list below is your checklist.

Your first two days. Day 1, morning: the walkthrough and Parts A and B (rules 1 to 13), about two hours; then open the claim microservice and find each class of the walkthrough in it. Day 1, afternoon: the API Design Developer Guide's walkthrough and Parts A and B. Day 2: Parts C and D of both guides, then your first ticket with a senior engineer. Parts E and F are not for reading through: they are the answers for the situations you will meet, one at a time, when you meet them.

The rules

One line per rule. The section with the same number has the situation, the code and the reasons.

Part A. Where a rule lives

1. Put a business rule in the entity method that makes the change, not in a service.
2. Change an entity's fields only through its own methods; no setters.
3. A service method only runs the steps: load, check the version, get facts, call the entity, save, audit.
4. Name classes, methods and errors with the business's own words.
5. Put a calculation that spans several objects in a domain service, and only when it has worked examples.
6. Give every ID and business value its own type; check its format once, where the value is created.

Part B. Aggregates

7. Keep a child inside the aggregate only if the parent's rules need it, it has a maximum count, it changes with the parent and it means nothing alone.
8. Store another aggregate's ID, never the object itself.
9. Save one aggregate per transaction.
10. Give every list inside an aggregate a maximum size.
11. Refuse a save that was made from an old version of the record.
12. Make every change safe to repeat: a repeated request adds or counts nothing twice.
13. Store money as an exact decimal and time in UTC, and count days in UK dates.

Part C. Features, microservices and events

14. Use another feature only through its application service, passing IDs and simple values.
15. Check permissions in the controller, never in the service or the entity.
16. Talk to another system only through one adapter class that translates its data into our types.
17. Call another microservice before you open the transaction, never inside it.
18. Retry a read; never retry a change.
19. Save an event in the same transaction as the change it reports, and send it afterwards.
20. Only add fields to a published event; to remove or rename one, publish a new version.
21. Write every listener so a message that arrives late, twice or out of order does no harm.
22. Change several microservices as a chain of steps joined by events, with a named step for failure.

Part D. Process

23. Get the design note approved before you code a feature with business rules.
24. Mark a feature Simple (data with no rules) or Rich (states and rules) in its design note.
25. Fix a mistake with a safe step today and the proper fix by a date.
26. Keep personal data out of logs, error messages and events.

Part E. Situations you will meet

27. When one button must save two aggregates, ask whether they are one; else react to an event; only then save both in one approved transaction.
28. Keep a rule across all records of a kind true with a unique index, and turn the index error into a business error.
29. Create an entity through a named factory method that checks the rules and raises the event; keep the constructor private.
30. Read for a screen through a query class on the collections; never build a screen out of aggregates.
31. Search and reports are queries too; give a slow one its own read collection filled by events.
32. Do not store on a parent what a child aggregate knows; read it when needed, or copy it by event and accept a short delay.
33. Never delete a business record; add a business action that marks it, with a reason.
34. Run a time-based rule from a scheduled job that calls the same entity method, one record per transaction.
35. When a process has its own steps and deadlines, give it its own aggregate.
36. Change an aggregate's shape behind a flag, with a migration that uses the entity's own methods.
37. Test the entity with no database, the service with a fake repository, and the repository alone against a real database.
38. Show two aggregates on one screen with two save buttons; one button that saves two things is a design smell to raise.

Terms

Term	Meaning
Microservice	A Spring Boot application deployed on its own, with its own DocumentDB database
Feature	One business area inside a microservice, in its own package. Example: `claims`, `notes`
Business action	One change the user asks for with one button, such as "close claim"

Term	Meaning
Entity	A business object with its own ID. Its details change; its identity does not. Example: `Claim`
Value object	A value with no ID, defined by its contents, never changed after it is created. Example: `Money`
Aggregate	A group of objects always loaded and saved together, as one unit. The top object is the aggregate root. In DocumentDB an aggregate is one document in one collection. Example: a `Claim` with its `EvidenceItem`s
Repository	The one class that loads and saves one kind of aggregate. Example: `ClaimRepository`, a Spring Data `MongoRepository`
Application service	A class with one method per business action. Each method runs that action's steps: load, check the version, get facts, call the entity, save, audit. Example: `ClaimService` with `open`, `addEvidence`, `close`. A class per action (`CloseClaim` with one method) is also fine; pick one shape per microservice
Domain service	A rare class for a calculation that belongs to no single entity. Example: `EntitlementCalculator`
Adapter	The one class that talks to another microservice or system. Example: `EligibilityClient`
Domain event	A record that something happened inside our microservice, named in the past tense. Example: `ClaimClosed`
Integration event	A versioned event that other microservices receive. Example: `ClaimClosedV1`
Outbox	A collection where an event is stored in the same transaction as the change it reports; a sender publishes it to the message broker afterwards
Inbox	A collection of processed message IDs, so a message delivered twice is handled once
Audit record	A line kept for ever after every change: who changed which record, how, and when. Written by the application service as its last step, never by the entity
Design group	The lead engineer, two or three senior engineers and the business experts
Design note	A one-page description of a feature that the design group approves before coding starts

The example

A **caseworker** records a **claim** for a **claimant**, adds **evidence** (documents such as payslips) and marks each item verified. The caseworker writes **notes**; a note can be marked "needs action". At the end, the caseworker closes the claim. The **eligibility** microservice says whether a person may claim. The **payment** microservice pays claimants and tells us when it has paid.

The business rules: a claim is created only if the claimant is eligible and the claim type is active. A claim has at most 10 evidence items, each verified once. A claim is closed only when all evidence is verified and no note needs action. A closed claim cannot be changed. A payment replaces the claim's last payment only if its payment date is later.

A claim and its evidence are one aggregate, so one document in the `claims` collection. Each note is its own aggregate in the `notes` collection and holds the claim's ID. Rule 7 explains why.

Closing a claim, end to end

This is the whole path of one business action, first when it succeeds and then when a rule refuses it. Every rule in the guide points back at one of these classes.

The controller receives the request and checks the permission. It does nothing else.

```
@RestController
public class ClaimController {
    @PostMapping("/v1/claims/{id}/close")
    @PreAuthorize("hasAuthority('CLOSE_CLAIM')")
    public ResponseEntity<Void> close(@PathVariable ClaimId id, @RequestBody @Valid CloseRequest
body) {
        claimService.close(id, body.version());
        return ResponseEntity.noContent().build();            // 204
    }
}

public record CloseRequest(@NotNull Long version) {}
```

Spring: `@PreAuthorize` with `@EnableMethodSecurity` refuses the call with `403` before the method body runs. `@PathVariable ClaimId` binds on its own, because `ClaimId` is a record with one `String` field (rule 6). `@Valid` turns a missing `version` into `400`.

The application service runs the steps. It holds no business rule.

```
@Service
public class ClaimService {
    @Transactional
    public void close(ClaimId id, long expectedVersion) {
        Claim claim = claims.findById(id).orElseThrow(() -> new ClaimNotFound(id));   // 1. load
        claim.requireVersion(expectedVersion);               // 2. the version the user saw
        int open = noteService.countNeedingAction(id);       // 3. a fact the claim does not hold
        claim.close(open, LocalDate.now(clock));             // 4. the entity decides
        claims.save(claim);                                  // 5. save; events go out with it
        audit.record("claim closed", id);                    // 6. audit
    }
}
```

Spring: `@Transactional` opens one DocumentDB transaction for the method, through the `MongoTransactionManager` bean in the setup below; without that bean the annotation does nothing.

The entity decides. All the rules for closing are in this one method.

```
@Document("claims")
public class Claim {
    @Id private ClaimId id;
    @Version private long version;
    private ClaimantId claimantId;
    private ClaimStatus status;
    private List<EvidenceItem> evidence = new ArrayList<>();   // stored inside the claim
document
    @Field(targetType = FieldType.STRING) private LocalDate closedOn;   // a date, stored as
"2026-10-31" (rule 13)
    @Transient private final List<DomainEvent> events = new ArrayList<>();
```

```
    public void requireVersion(long expected) {              // step 2 of every change (rule 11)
        if (version != expected) throw new VersionConflict(id, expected, version);
    }

    public void close(int notesNeedingAction, LocalDate today) {
        if (status == CLOSED) throw new ClaimAlreadyClosed(id);
        if (evidence.stream().anyMatch(e -> !e.isVerified())) throw new EvidenceNotVerified(id);
        if (notesNeedingAction > 0) throw new ClaimHasOpenActions(id, notesNeedingAction);
        status = CLOSED;
        closedOn = today;
        events.add(new ClaimClosed(id, today));
    }

    @DomainEvents Collection<DomainEvent> events() { return List.copyOf(events); }
    @AfterDomainEventPublication void clearEvents() { events.clear(); }
}
```

Spring: `@Document` names the collection; `@Version` makes `save` fail with `OptimisticLockingFailureException` when the stored version is not the loaded one; `@DomainEvents` publishes the entity's events when the repository saves it; `@Transient` keeps the event list out of the document.

The repository loads and saves the whole aggregate. It is an interface; Spring Data writes the implementation.

```
public interface ClaimRepository extends MongoRepository<Claim, ClaimId> {}
```

The outbox listener stores each event beside the claim, in the same transaction. A sender publishes the stored events after the commit.

```
@Component
public class Outbox {
    @EventListener                                           // same thread, same transaction as
the save
    public void store(DomainEvent event) {
        if (!event.hasPublishedShape()) return;              // internal-only events stay inside
        mongo.insert(OutboxEntry.of(event), "outbox");       // ClaimClosed becomes a
ClaimClosedV1 entry
    }

    @Scheduled(fixedDelay = 1000)                            // on more than one instance, under
the job lock (rule 34)
    public void send() {
        for (OutboxEntry entry : mongo.find(query(where("sentAt").isNull()), OutboxEntry.class,
"outbox")) {
            broker.publish(entry.topic(), entry.payload());
            mongo.updateFirst(query(where("_id").is(entry.id())), update("sentAt",
Instant.now(clock)), "outbox");
        }
    }
}
```

Spring: `@EventListener` receives the events `@DomainEvents` published, synchronously, so the `insert` runs inside the service's transaction; `@Scheduled` with `@EnableScheduling` runs the sender; `MongoTemplate` does the writes.

The published event carries IDs, not personal data.

```
public record ClaimClosedV1(String eventId, String claimId, LocalDate closedOn) {}
```

When a rule refuses

Now the claim has two notes that need action. `Claim.close` throws `ClaimHasOpenActions` at its third line. Nothing catches it on the way out: not the service, not the controller. `@Transactional` rolls the transaction back on the unchecked exception, so nothing is saved, no outbox entry is stored and no audit record is written. The exception reaches one shared error handler at the HTTP edge, which turns it into the answer:

```
public abstract class DomainError extends RuntimeException {   // unchecked, so the transaction
rolls back
    protected DomainError(String message) { super(message); }
    public abstract String code();
}
public abstract class RuleRefused extends DomainError { ... }    // a rule said no: 422
public abstract class NotFound extends DomainError { ... }       // the record does not exist:
404
public final class VersionConflict extends DomainError { ... }   // the version is old: 409
public final class ClaimHasOpenActions extends RuleRefused {
    public ClaimHasOpenActions(ClaimId id, int notes) { super("Claim " + id.value() + " has " +
notes + " notes that need action"); }
    public String code() { return "CLAIM_HAS_OPEN_ACTIONS"; }
}

@RestControllerAdvice
public class ApiErrors extends ResponseEntityExceptionHandler {   // the one error handler of the
microservice
    @ExceptionHandler(RuleRefused.class)
    ProblemDetail ruleRefused(RuleRefused e) {
        return problem(HttpStatus.UNPROCESSABLE_CONTENT, "Business rule violated", e.code(),
e.getMessage());
    }
    @ExceptionHandler({VersionConflict.class, OptimisticLockingFailureException.class})
    ProblemDetail conflict(Exception e) {
        return problem(HttpStatus.CONFLICT, "Record changed", "CONCURRENT_UPDATE",
e.getMessage());
    }
    @ExceptionHandler(NotFound.class)
    ProblemDetail notFound(NotFound e) {
        return problem(HttpStatus.NOT_FOUND, "Not found", e.code(), e.getMessage());
    }

    private ProblemDetail problem(HttpStatus status, String title, String code, String detail) {
        ProblemDetail p = ProblemDetail.forStatusAndDetail(status, detail);
        p.setTitle(title);
        p.setProperty("code", code);
        p.setProperty("traceId", Trace.current());      // the trace ID a filter took from the
traceparent header
        return p;
    }
}
```

```
422 Unprocessable Content
Content-Type: application/problem+json
{ "status": 422, "title": "Business rule violated", "code": "CLAIM_HAS_OPEN_ACTIONS",
  "detail": "Claim 9f1c has 2 notes that need action", "traceId": "4bf92f35..." }
```

Spring: `@RestControllerAdvice` with `@ExceptionHandler` runs for every controller; `ProblemDetail` is Spring MVC's RFC 9457 body and `setProperty` adds our `code` and `traceId`. `@Transactional` rolls back on any `RuntimeException`, so every domain error is one.

Three kinds of error are thrown on purpose: not found, old version, rule refused. Anything else is a bug and answers `500` with the trace ID, never the stack trace. The same class also answers `400`, `403` and `503`; the API Design Developer Guide, rule 26, shows those handlers.

When the same exception happens in a message listener, there is no HTTP answer. The transaction rolls back, the message is not marked processed, and the broker delivers it again. A rule that keeps refusing sends the message to the dead-letter queue (rule 21).

One-time setup for the stack

```
@Configuration
@EnableScheduling @EnableMethodSecurity
public class AppConfig {
    @Bean MongoTransactionManager transactionManager(MongoDatabaseFactory db) { return new
MongoTransactionManager(db); }
    @Bean Clock clock() { return Clock.systemUTC(); }
    @Bean MongoCustomConversions conversions() {
        return new MongoCustomConversions(List.of(new ClaimIdToString(), new StringToClaimId(),
/* one pair per ID type */));
    }

    @Bean JwtAuthenticationConverter jwtAuthorities() {     // the gateway's token carries
"permissions": ["CLOSE_CLAIM", ...]
        JwtGrantedAuthoritiesConverter claims = new JwtGrantedAuthoritiesConverter();
        claims.setAuthoritiesClaimName("permissions");
        claims.setAuthorityPrefix("");                       // so hasAuthority('CLOSE_CLAIM')
matches as written
        JwtAuthenticationConverter converter = new JwtAuthenticationConverter();
        converter.setJwtGrantedAuthoritiesConverter(claims);
        return converter;
    }
}
```

```
spring.mongodb.uri=mongodb://...docdb.amazonaws.com:27017/claims?
tls=true&replicaSet=rs0&retryWrites=false
spring.data.mongodb.auto-index-creation=false        # indexes are created by a migration, never
at startup
spring.security.oauth2.resourceserver.jwt.issuer-uri=https://gateway.example.gov.uk   # Boot
builds the JWT decoder from it
```

Spring: `MongoTransactionManager` turns `@Transactional` on, and Boot then adds the `TransactionTemplate` bean of rule 9. `MongoCustomConversions` stores each ID value object as a plain string. With `issuer-uri` set, Boot's default `SecurityFilterChain` demands a valid token on every request, and `JwtAuthenticationConverter` makes `hasAuthority('CLOSE_CLAIM')` see the token's

`permissions` claim; without it Spring reads only `scope`, as `SCOPE_...`, and every check fails. DocumentDB before engine 8.0.2 rejects retryable writes, hence `retryWrites=false`. Index creation stays off: a DocumentDB index build blocks the collection, so a migration does it at a planned time.

Part A. Where a rule lives

1. Put a business rule in the entity method that makes the change, not in a service

You are adding "close claim". The rule: a claim closes only when every evidence item is verified and no note needs action.

Bad — the rule sits in the service:

```
@Service
public class ClaimService {
    @Transactional
    public void close(ClaimId id) {
        Claim claim = claims.findById(id).orElseThrow();
        if (claim.getStatus() == CLOSED) throw new IllegalStateException();
        if (!claim.getEvidence().stream().allMatch(EvidenceItem::isVerified)) throw new
IllegalStateException();
        if (noteService.countNeedingAction(id) > 0) throw new IllegalStateException();
        claim.setStatus(CLOSED);
        claims.save(claim);
    }
}
```

Next month someone adds "close claims from the nightly job" and copies these three `if`s. Then one copy changes and the other does not.

Good — the rule sits in `Claim.close`. The service fetches the one fact the claim does not hold (the note count) and passes it in:

```
public class Claim {
    public void close(int notesNeedingAction, LocalDate today) {
        if (status == CLOSED) throw new ClaimAlreadyClosed(id);
        if (evidence.stream().anyMatch(e -> !e.isVerified())) throw new EvidenceNotVerified(id);
        if (notesNeedingAction > 0) throw new ClaimHasOpenActions(id, notesNeedingAction);
        status = CLOSED;
        closedOn = today;
        events.add(new ClaimClosed(id, today));
    }
}

@Service
public class ClaimService {
    @Transactional
    public void close(ClaimId id, long expectedVersion) {
        Claim claim = claims.findById(id).orElseThrow(() -> new ClaimNotFound(id));
        claim.requireVersion(expectedVersion);
        int open = noteService.countNeedingAction(id);
        claim.close(open, LocalDate.now(clock));
        claims.save(claim);
```

```
        audit.record("claim closed", id);
    }
}
```

Nobody can call `close` without supplying the note count, so the check cannot be forgotten. The nightly job calls the same `Claim.close`.

2. Change an entity's fields only through its own methods; no setters

You need to mark an evidence item verified.

Bad — a setter:

```
item.setVerified(true);
item.setVerifiedOn(today);
```

Nothing checks that the item was not verified already, and no `EvidenceVerified` event is recorded. Any class anywhere can do this.

Good — a method with the business name, on the entity:

```
public class Claim {
    public void verifyEvidence(EvidenceId evidenceId, LocalDate today) {
        if (status == CLOSED) throw new ClaimAlreadyClosed(id);
        EvidenceItem item = find(evidenceId);
        item.markVerified(today);                          // refuses a second time
        events.add(new EvidenceVerified(id, evidenceId));
    }
}
```

All fields are private, and a build check fails on any public setter in an entity. A method checks the rules and records the event; a setter skips both.

Spring: Spring Data MongoDB reads and writes private fields directly, so an entity needs no getters or setters for persistence.

3. A service method only runs the steps: load, check the version, get facts, call the entity, save, audit

You are writing `ClaimService.verifyEvidence`.

Bad — the service makes decisions:

```
@Transactional
public void verifyEvidence(ClaimId id, EvidenceId evidenceId) {
    Claim claim = claims.findById(id).orElseThrow();
    if (claim.getStatus() == CLOSED) throw new IllegalStateException();
    EvidenceItem item = claim.getEvidence().stream().filter(e ->
e.id().equals(evidenceId)).findFirst().orElseThrow();
    if (item.isVerified()) throw new IllegalStateException();
```

```
    item.setVerified(true);
    claims.save(claim);
}
```

"Verified once" sits in the service, which is rule 1's problem. The shape is the second problem: this method has no version check and no audit, and nobody notices, because every service method in the codebase looks different.

Good — the same six steps as `close`, and nothing else:

```
@Transactional
public void verifyEvidence(ClaimId id, long expectedVersion, EvidenceId evidenceId) {
    Claim claim = claims.findById(id).orElseThrow(() -> new ClaimNotFound(id));   // 1
    claim.requireVersion(expectedVersion);                    // 2
                                                              // 3: no outside facts needed
    claim.verifyEvidence(evidenceId, LocalDate.now(clock));  // 4: the entity checks the rules
(rule 2)
    claims.save(claim);                                       // 5
    audit.record("evidence verified", id);                    // 6
}
```

A service method contains no `if` about status, amounts, dates or counts; such an `if` belongs in the entity (rule 1) or a value object (rule 6). The transaction covers steps 1 to 6; a call to another system happens before it opens (rule 17). Take the time from the `Clock` bean, so tests can set the date. The fixed order is a convention we chose: when every action has the same six steps in the same order, a missing version check or audit stands out in review.

Spring: `@Transactional` on a public method of a `@Service` bean. A call from another method of the same class bypasses it, so keep each business action in its own public method, or start the transaction with `TransactionTemplate` (rule 9).

4. Name classes, methods and errors with the business's own words

The business says "close a claim" and "a claim has open actions".

Bad:

```
claim.updateStatus(3);
throw new ValidationException("invalid state");
```

A caseworker reading the log cannot tell what happened, and a developer must look up what 3 means.

Good:

```
claim.close(openActions, today);
throw new ClaimHasOpenActions(id, openActions);
```

The design note, the code, the error message and the conversation with the business then use the same words. The shared `@RestControllerAdvice` turns `ClaimHasOpenActions` into `422` with the code `CLAIM_HAS_OPEN_ACTIONS` ("When a rule refuses", in the walkthrough). The entity never knows about HTTP.

5. Put a calculation that spans several objects in a domain service, and only when it has worked examples

You must calculate a claimant's weekly entitlement from the claim type, the rate table and the number of children.

Bad — a "policy" class that holds one `if`:

```
@Component
public class ClaimClosurePolicy {
    public boolean canClose(Claim claim) {
        return claim.evidenceAllVerified();
    }
}
```

This is one `if` about closing, so it belongs in `Claim.close` (rule 1); a helper bean is one more place to look. A domain service is for the opposite case: a calculation over several objects that would swamp the entity.

Good — a stateless calculation with a table of worked examples in the design note:

```
public class EntitlementCalculator {                          // a plain class; not a bean,
nothing injected
    public Money weeklyEntitlement(ClaimType type, RateTable rates, int children) {
        Money base = rates.baseFor(type);
        return base.plus(rates.perChild().times(children));
    }
}
```

Claim type	Children	Weekly entitlement
Standard	0	£84.80
Standard	2	£146.20

A domain service never loads or saves. If you cannot write a table like this, the logic is one or two `if`s and belongs in the entity (rule 1).

6. Give every ID and every business value its own type, and check its format once, where the value is created

An amount must not be negative. A file name must not be blank. A claimant ID must never be passed where a claim ID is expected.

Bad — checks sprinkled along the way, and bare values:

```
// in the controller
if (body.amount() < 0) return ResponseEntity.badRequest().build();
// in the service
if (amount < 0) throw new IllegalArgumentException();
// in the entity
if (amount < 0) throw new IllegalArgumentException();
```

```
public void close(String claimId, String claimantId, BigDecimal amount) { ... }   // three bare
values
```

Three copies of the check, and the import job has none. And `close(claimantId, claimId, ...)` compiles, with the two IDs swapped.

Good — every ID and every business value is its own small type, and it refuses to exist with a bad value:

```
public record ClaimId(String value) {
    public ClaimId { if (!value.matches("[0-9a-f]{4,32}")) throw new InvalidClaimId(value); }
}

public record Money(BigDecimal amount) {
    public Money {
        if (amount.signum() < 0) throw new NegativeAmount(amount);
        if (amount.scale() > 2) throw new TooManyDecimalPlaces(amount);
        amount = amount.setScale(2);
    }
}

public void close(ClaimId id, int notesNeedingAction, LocalDate today) { ... }    // the compiler
checks the types
```

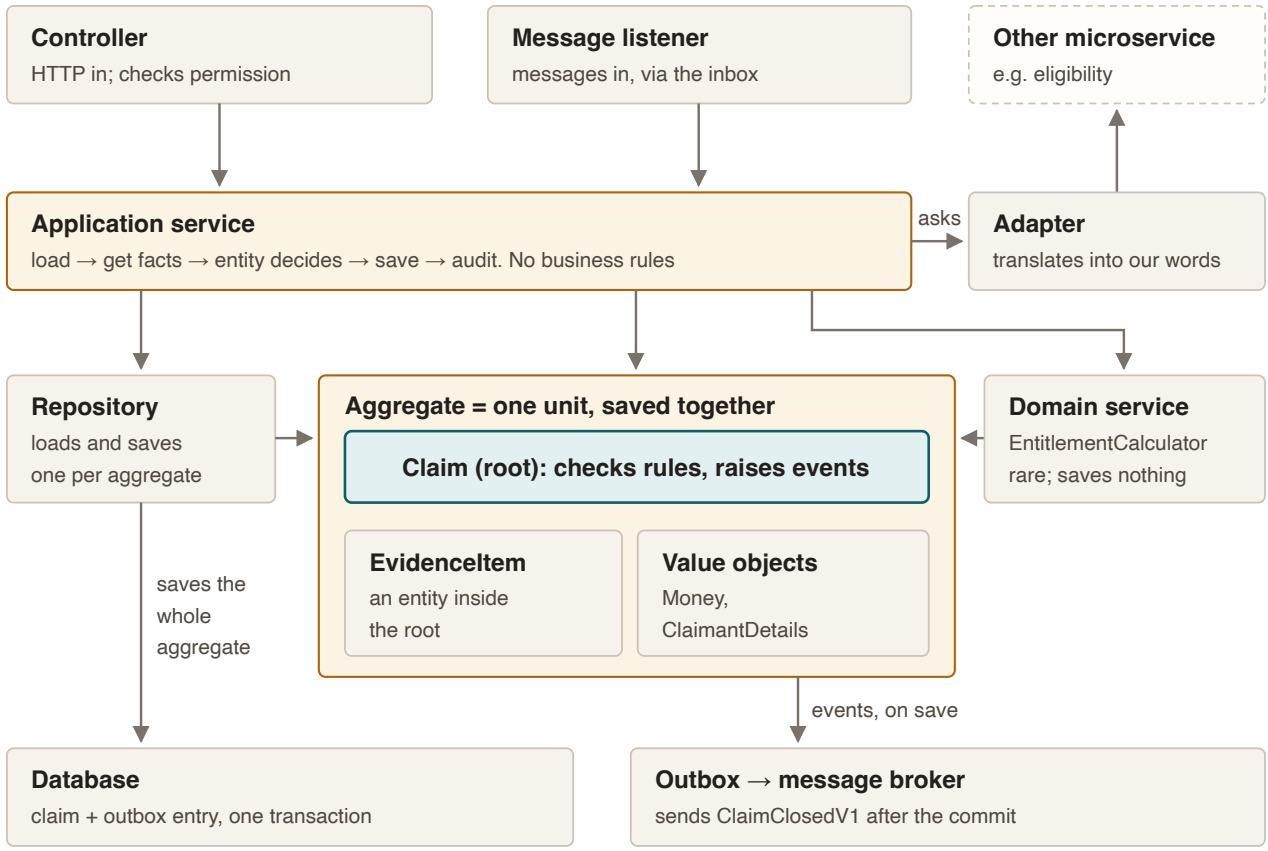
```
@WritingConverter public class ClaimIdToString implements Converter<ClaimId, String> { public
String convert(ClaimId id) { return id.value(); } }
@ReadingConverter public class StringToClaimId implements Converter<String, ClaimId> { public
ClaimId convert(String s) { return new ClaimId(s); } }
```

`ClaimId`, `ClaimantId`, `NoteId`, `Money`, `NoteText`, `FileName`, `Nino`: never a bare `String` or `BigDecimal` inside the microservice. A bare string arrives only at the edges (a request body, a message, a document) and becomes a type there. Once a `Money` exists, it is valid, everywhere, and two IDs cannot be swapped by mistake.

Spring: a `@WritingConverter`/`@ReadingConverter` pair, registered in `MongoCustomConversions`, stores an ID as a plain string in the document. Spring MVC needs no converter to bind `@PathVariable ClaimId`: a record with one `String` field has a constructor Spring's conversion calls itself. `Money` is stored as a nested object with a `Decimal128`, exact.

Where does this code go? Stop at the first yes

1. Does it decide whether the entity may change, from its current state? Entity method (rule 1).
2. Is it a calculation across several things, with worked examples? Domain service (rule 5).
3. Does it load, save, call another microservice, start a transaction or write audit? Application service (rule 3).
4. Does it check the format of one value? Value object (rule 6), or the factory method that creates the entity (rule 29).
5. Does it check a code from another feature? That feature's application service (rule 14).



How the building blocks handle one request: the application service runs the steps, and the aggregate decides. A controller or a message listener calls the application service. It asks other microservices through an adapter, loads and saves through the repository, and uses a domain service for a calculation. The Claim root checks the rules and raises events. The repository saves the whole aggregate, with its outbox entry, in one transaction, and the outbox sends the events to the message broker after the commit.

Part B. Aggregates

7. Keep a child inside the aggregate only when the parent's rules need its details, it has a maximum count, it changes with the parent and it means nothing alone; otherwise give it its own aggregate

You are deciding where evidence items and notes live.

Bad — everything a claim has, inside the claim document:

```
@Document("claims")
public class Claim {
    private List<EvidenceItem> evidence = new ArrayList<>();
    private List<Note> notes = new ArrayList<>();                // grows for years
    private List<Payment> payments = new ArrayList<>();          // grows for years
}
```

Two caseworkers add a note to the same claim at the same time. The whole claim is one document, so the second save fails on the version (rule 11). After three years the document holds 4,000 notes and the save fails

on DocumentDB's 16 MiB document limit.

Good — ask the heading's four questions about each child. Four times yes: inside the document. Any no: its own collection, holding the parent's ID.

Child	Rule needs details?	Maximum?	Changes with parent?	Meaningless alone?	Decision
Evidence item	Yes: closing needs every item verified	10	Yes	Yes	Inside the claim document
Note	No: closing needs only a count	None	No: two caseworkers at once	No	Own collection `notes`
Payment	No: only the latest	None	No: the payment microservice sends them	No	Own collection `payments`

```
@Document("claims")
public class Claim {
    private List<EvidenceItem> evidence = new ArrayList<>();   // inside: an array in the claim
document
}

@Document("notes")
public class Note {
    @Id private NoteId id;
    @Version private long version;
    @Indexed private ClaimId claimId;                          // points at the parent
    private NoteText text;
    private boolean needsAction;
}
```

Inside one document a rule is always true, because the document is written atomically. Between collections a rule is true only at the moment it is checked, which is why `close` asks for the note count at that moment. Copying the old system's tables one to one as aggregates is the most common way to get this wrong.

Spring: a nested object or list in an entity is stored inside the same document; a separate `@Document` class is a separate collection. `@Indexed` on `claimId` names the index the migration creates.

8. Store another aggregate's ID, never the object itself

A note belongs to a claim.

Bad:

```
@Document("notes")
public class Note {
    @DocumentReference private Claim claim;                    // or @DBRef
}
```

Loading a note now runs a second query for the claim and its evidence; a list of 20 notes runs 21 queries. `@DBRef` is worse: DocumentDB does not support DBRefs at all.

Good:

```
@Document("notes")
public class Note {
    private final ClaimId claimId;
}
```

When the note feature needs something about the claim, it asks `ClaimService` with the ID (rule 14).

Spring: `@DocumentReference` stores the ID but loads the object on read, one query per note; a plain `ClaimId` field stores the ID and loads nothing.

9. Save one aggregate per transaction

A nightly job must mark 10,000 claims as "review due".

Bad:

```
@Transactional
public void markReviewDue(LocalDate today) {
    for (Claim claim : claims.findByReviewDueOnBefore(today)) {
        claim.markReviewDue(today);
        claims.save(claim);
    }
}
```

One DocumentDB transaction has a one-minute limit and a 32 MB log; this one hits both, and if claim 9,999 fails nothing is saved.

Good — one save per claim, each atomic on its own, and skip claims already done:

```
public void markReviewDue(LocalDate today) {
    for (ClaimId id : claims.findIdsByReviewDueOnBefore(today)) {       // a query that returns
only IDs (rule 30)
        tx.executeWithoutResult(status -> markOne(id, today));          // one transaction per
claim
    }
}

private void markOne(ClaimId id, LocalDate today) {
    Claim claim = claims.findById(id).orElseThrow(() -> new ClaimNotFound(id));
    if (claim.isReviewDue()) return;                          // already done; safe to rerun
    claim.markReviewDue(today);
    claims.save(claim);
    audit.record("review due", id);
}
```

Saving two aggregates in one transaction needs an approved exception in the design note. Example: creating a claim together with its first note (rule 27).

Spring: a single-document `save` is atomic in DocumentDB without any transaction; the transaction here covers the outbox entry saved beside the claim. `tx` is Boot's `TransactionTemplate` bean. It starts a

transaction in the middle of a method, which `@Transactional` cannot: a call from inside the same class never passes through the proxy that starts one.

10. Give every list inside an aggregate a maximum size

A claim holds evidence items.

Bad:

```
public void addEvidence(FileName fileName, LocalDate today) {
    evidence.add(new EvidenceItem(fileName, today));
}
```

Nothing stops a claim with 600 items. Its page times out, and the document heads for DocumentDB's 16 MiB limit, in production, for one user.

Good — the design note says "at most 10", and the entity refuses the eleventh:

```
private static final int MAX_EVIDENCE = 10;

public EvidenceId addEvidence(FileName fileName, FileReference file, LocalDate today) {
    if (status == CLOSED) throw new ClaimAlreadyClosed(id);
    if (evidence.size() >= MAX_EVIDENCE) throw new TooMuchEvidence(id, MAX_EVIDENCE);
    EvidenceItem item = EvidenceItem.create(fileName, file, today);
    evidence.add(item);
    events.add(new EvidenceAdded(id, item.id()));             // the ID, not the file name (rule
26)
    return item.id();
}
```

A list that cannot have a maximum is not a child; it is its own collection (rule 7). The file itself is never in the document: the service stores it in the object store first, and the evidence item holds its reference (API guide, rule 34).

11. Refuse a save that was made from an old version of the record

Two caseworkers open the same claim. One adds evidence and saves. The other, still looking at the old screen, closes the claim.

Bad — the second save wins:

```
Claim claim = claims.findById(id).orElseThrow();
claim.close(open, today);
mongo.save(claim);                                          // no version field: overwrites the
new evidence
```

The first caseworker's evidence is gone and nobody is told.

Good — every change carries the version the user saw, the entity refuses an old one, and the document's `@Version` refuses it again in the database:

```
@Document("claims")
public class Claim {
    @Version private long version;

    public void requireVersion(long expected) {
        if (version != expected) throw new VersionConflict(id, expected, version);
    }
}
```

`requireVersion`, step 2 of every change, gives the friendly error; `@Version` repeats the check in the database, because two saves in the same millisecond can both pass the entity's. The error handler maps both to `409`, and the screen says "changed by someone else; reload". Never remove either check, or update one field with `MongoTemplate` to get around them.

Spring: `@Version` makes `save` send `{_id, version: 7}` as the filter and `version: 8` in the update; no match means `OptimisticLockingFailureException`.

12. Make every change safe to repeat: a repeated request must not add or count anything twice

The payment microservice sends `PaymentMadeV1`. The broker delivers it twice.

Bad:

```
public void recordPayment(Money amount) {
    totalPaid = totalPaid.plus(amount);                    // twice is wrong
}
```

The broker delivers the event twice and the total is doubled. The inbox (rule 21) stops the second delivery; this rule adds the entity's own guard, for the day a replay or a migration bypasses the inbox.

Good — the entity keeps what it already has, and ignores a repeat:

```
public void recordPayment(PaymentId paymentId, Money amount, LocalDate paidOn) {
    if (lastPayment != null && !paidOn.isAfter(lastPayment.paidOn())) return;   // older or same:
ignore
    lastPayment = new Payment(paymentId, amount, paidOn);
    events.add(new PaymentRecorded(id, paymentId));
}
```

A request is repeated after a timeout, because the caller cannot tell whether the first one landed. Three things together make a repeat harmless: the version check for changes from a screen (rule 11), an idempotency key for create requests (API guide, rule 4), and the inbox for messages (rule 21). The entity's own check is the fourth, and the only one that holds on every path.

13. Store money as an exact decimal and time in UTC, and count days in UK dates

A claim can be reopened within 28 days of closing.

Bad:

```
private double amount;                                        // 0.1 + 0.2 = 0.30000000000000004
private LocalDate closedOn;                                   // Spring Data stores it as a Date
at local midnight
Instant deadline = closedAt.plus(28 * 24, ChronoUnit.HOURS); // an hour out after the clocks
change
```

A `LocalDate` of 1 June written from a machine on UK time is stored as `2026-05-31T23:00:00Z` and reads back as 31 May on a server set to UTC.

Good:

```
public record Money(BigDecimal amount) { ... }              // stored as Decimal128, exact (rule
6)

private Instant closedAt;                                   // stored as a UTC date
@Field(targetType = FieldType.STRING) private LocalDate closedOn;   // stored as "2026-06-01"

static final ZoneId UK = ZoneId.of("Europe/London");
LocalDate closedOn = closedAt.atZone(UK).toLocalDate();
LocalDate deadline = closedOn.plusDays(28);                 // a date, in UK time
```

Floating point gives penny errors. Local timestamps are an hour wrong twice a year. A rule that counts days works in dates, not hours.

Spring: on Boot 4 a `BigDecimal` is stored as `Decimal128` by default (`spring.data.mongodb.representation.big-decimal`); `@Field(targetType = FieldType.STRING)` stores a `LocalDate` as an ISO string instead of a shifted `Date`; `Instant` is always UTC.

Part C. Features, microservices and events

14. Use another feature only through its application service, passing IDs and simple values

The claim feature needs to know how many notes need action.

Bad — reaching into the other feature's storage:

```
@Service
public class ClaimService {
    private final NoteRepository noteRepository;            // another feature's repository
    ...
    long open = noteRepository.findByClaimId(id).stream().filter(Note::needsAction).count();
}
```

When the notes team changes how notes are stored, the claim feature breaks. The "needs action" rule now lives in two features.

Good — ask the other feature's application service for the answer:

```
@Service
public class ClaimService {
    private final NoteService noteService;                  // the other feature's front door
    ...
    int open = noteService.countNeedingAction(id);
}
```

Pass IDs, numbers and small records, never entities or repositories. Calls go one way: claims may call notes, so notes must not call claims; notes use events for the other direction (rule 19). A build check enforces this.

Spring: both are `@Service` beans injected by constructor; a repository bean is package-private to its feature, so the other feature cannot inject it.

15. Check permissions in the controller, never in the service or the entity

Only a caseworker with `CLOSE_CLAIM` may close a claim.

Bad:

```
public class Claim {
    public void close(Authentication user, int open, LocalDate today) {
        if (!user.getAuthorities().contains("CASEWORKER")) throw new AccessDeniedException("no");
        ...
    }
}
```

A message listener also closes claims (when the payment microservice says the last payment was made). There is no user there, so someone passes a fake system user, and the check means nothing.

Good — the controller knows the user, so the controller checks:

```
@PostMapping("/v1/claims/{id}/close")
@PreAuthorize("hasAuthority('CLOSE_CLAIM')")
public ResponseEntity<Void> close(@PathVariable ClaimId id, @RequestBody @Valid CloseRequest
body) {
    claimService.close(id, body.version());
    return ResponseEntity.noContent().build();
}

@WebMvcTest(ClaimController.class)
class ClaimControllerTest {
    @Test @WithMockUser(authorities = "VIEW_CLAIM")
    void refusesWithoutPermission() throws Exception {      // csrf(): a MockMvc call carries no
bearer token, so the CSRF filter applies

mvc.perform(post("/v1/claims/9f1c/close").with(csrf()).contentType(APPLICATION_JSON).content("
{\"version\":7}"))
           .andExpect(status().isForbidden());
    }
}
```

A test proves that a user without the permission gets `403`. Application services and entities never see a user.

Spring: `@PreAuthorize` on the controller method, enabled by `@EnableMethodSecurity`; the authority comes from the JWT the gateway sends. `@WebMvcTest` with `@WithMockUser` tests the check without a server.

16. Talk to another system only through one adapter class that translates its data into our types

Creating a claim needs the eligibility microservice's answer.

Bad — their data leaks into our entity:

```
Map<String, Object> response = restClient.get().uri("/eligibility/" +
nino).retrieve().body(Map.class);
if ("ELIGIBLE".equals(response.get("status"))) { ... }       // their word, inside our code
```

When the eligibility team renames `status` to `outcome`, we hunt through our whole codebase.

Good — one adapter class, returning our own type and our own errors:

```
@Component
public class EligibilityClient {
    private final RestClient http;                          // built once, with time limits, in
a @Configuration

    public Eligibility check(ClaimantId claimantId) {
        EligibilityResponse theirs;
        try {
            theirs = http.get().uri("/v1/eligibility/{id}",
claimantId.value()).retrieve().body(EligibilityResponse.class);
        } catch (ResourceAccessException e) {               // no connection or no answer: our
error, not theirs (rule 18)
            throw new EligibilityUnavailable(claimantId);
        }
        return switch (theirs.status()) {
            case "ELIGIBLE" -> Eligibility.ELIGIBLE;
            case "NOT_ELIGIBLE" -> Eligibility.NOT_ELIGIBLE;
            default -> throw new EligibilityUnknown(claimantId);
        };
    }
}
```

Nothing from the other system appears outside its adapter. Nobody reads another microservice's database.

Spring: `RestClient` is Spring's HTTP client; `EligibilityResponse` is a record Spring deserialises from their JSON and that only this class sees. The adapter is tested against WireMock playing the eligibility microservice (rule 37).

17. Call another microservice before you open the transaction, never inside it

Creating a claim asks eligibility, then saves.

Bad:

```
@Transactional
public ClaimId open(ClaimantId claimantId, ClaimType type) {
    Eligibility eligibility = eligibilityClient.check(claimantId);   // a network call, inside
the transaction
    Claim claim = Claim.open(claimantId, type, eligibility, LocalDate.now(clock));
    claims.save(claim);
    return claim.id();
}
```

Eligibility takes four seconds today. For four seconds the transaction holds its session and its locks, and it now fails whenever the network does. Had the outside call made a change, a failed commit could not undo it.

Good — the call first, then the transaction:

```
public ClaimId open(ClaimantId claimantId, ClaimType type) {
    Eligibility eligibility = eligibilityClient.check(claimantId);   // outside
    return tx.execute(status -> {                                    // the transaction starts
here (rule 9)
        Claim claim = Claim.open(claimantId, type, eligibility, LocalDate.now(clock));
        claims.save(claim);
        audit.record("claim opened", claim.id());
        return claim.id();
    });
}
```

Get every outside fact first, then open the transaction and save. The object store is another system too: `addEvidence` stores the file, then opens the transaction to save its reference (rule 10).

Spring: `@Transactional` would start at the method call, network call included; `TransactionTemplate` starts where the lambda does.

18. Retry a read; never retry a change

The eligibility check timed out.

Bad — retry everything the same way:

```
@Retryable                                                  // on the adapter's change method
public void changeAddress(ClaimantId claimantId, Address newAddress) { ... }
```

The first call reached the other system and changed the address; only the reply was lost. The retry changes it again, and the audit trail there now shows two changes we cannot undo.

Good — reads may retry; a change is sent once:

```
@Retryable(maxRetries = 2, includes = ResourceAccessException.class)   // on the adapter's read
method (rule 16): safe
public Eligibility check(ClaimantId claimantId) { ... }

try {
    addressClient.changeAddress(claimantId, newAddress);                        // a change:
```

```
once
} catch (ResourceAccessException e) {
    if (e.getCause() instanceof ConnectException) throw new AddressNotChanged(claimantId);   //
never reached them
    throw new AddressChangeUnknown(claimantId);           // reached them, reply lost; tell the
user to check
}
```

Stop calling a system that keeps failing, for a short time (a circuit breaker). A business action that sends a change to another system makes exactly one call and saves nothing else of its own.

Spring: `@Retryable`, from Spring Framework 7 and turned on by `@EnableResilientMethods`, retries the method; `maxRetries` counts retries, not calls. `RestClient` throws `ResourceAccessException` when the connection fails or times out, and the cause says which. Spring has no circuit breaker of its own.

19. Save an event in the same transaction as the change it reports, and send it afterwards

Closing a claim must tell the payment microservice.

Bad — publish from the service:

```
claims.save(claim);
broker.publish("claims", new ClaimClosedV1(...));         // the broker is down: saved, never
sent
```

Or the other way round: published first, then the save fails. Payments stop for a claim that is still open.

Good — the entity records the event; an outbox entry is stored in the same transaction; a scheduled sender publishes it (the `Outbox` class in the walkthrough):

```
// in Claim.close
events.add(new ClaimClosed(id, today));

// Claim: Spring Data publishes these when claims.save(claim) runs
@DomainEvents Collection<DomainEvent> events() { return List.copyOf(events); }
@AfterDomainEventPublication void clearEvents() { events.clear(); }

// Outbox: same thread, same transaction
@EventListener
public void store(DomainEvent event) { mongo.insert(OutboxEntry.of(event), "outbox"); }

// Outbox: after the commit, every second, retried until the broker takes it
@Scheduled(fixedDelay = 1000)
public void send() { ... }
```

A saved change always sends its event, and an unsaved change never does. Nobody publishes any other way.

Spring: `@DomainEvents` and `@AfterDomainEventPublication` on the aggregate root, `@EventListener` on the outbox (synchronous, so inside the transaction), `@Scheduled` for the sender. Spring Data publishes the events only on a repository `save`, never on a `MongoTemplate` write. Only the outbox and a read collection (rule 31) listen to the entity's events directly; every other listener, ours included, gets them from the broker (rule 27).

20. Only add fields to a published event; to remove or rename one, publish a new version

The payment team reads `ClaimClosedV1.closedOn`. You want to call it `closureDate`.

Bad:

```
public record ClaimClosedV1(String eventId, String claimId, LocalDate closureDate) {}   //
renamed in place
```

The payment microservice's listener reads `closedOn`, gets null, and stops payments on the wrong date.

Good — add freely; for a rename, add a version and send both until every receiver has moved:

```
public record ClaimClosedV1(String eventId, String claimId, LocalDate closedOn, String
closedByUserId) {}   // a new field: fine
public record ClaimClosedV2(String eventId, String claimId, LocalDate closureDate, String
closedByUserId) {} // sent beside V1
```

Every integration event has a written schema. Tell the receiving teams before the release. A domain event, used only inside our microservice, can change at any time.

21. Write every message listener so a message that arrives late, twice or out of order does no harm

The payment microservice sends `PaymentMadeV1`. The broker may deliver it twice, or deliver March's payment after April's.

Bad:

```
// called by the message listener for each PaymentMadeV1
public void on(PaymentMadeV1 event) {
    Claim claim = claims.findById(new ClaimId(event.claimId())).orElseThrow();
    claim.setLastPayment(event.amount(), event.paidOn());    // the last one to arrive wins
    claims.save(claim);
}
```

The second copy of March's payment is saved again, and when April's arrives first, March's overwrites it.

Good — an inbox skips repeats, and the entity ignores anything older than what it holds:

```
@Component
public class Inbox {                                          // one per microservice
    public boolean seen(String eventId) { return mongo.exists(query(where("_id").is(eventId)),
"inbox"); }
    public void record(String eventId) { mongo.insert(new InboxEntry(eventId,
Instant.now(clock)), "inbox"); }
}
public record InboxEntry(@Id String eventId, Instant processedAt) {}

// called by the message listener for each PaymentMadeV1
@Transactional
public void on(PaymentMadeV1 event) {
    if (inbox.seen(event.eventId())) return;                  // already processed: skip
```

```
    inbox.record(event.eventId());                          // _id is unique: a race loses here
    claimService.recordPayment(                             // translate their strings into our
types
        new ClaimId(event.claimId()), new PaymentId(event.paymentId()),
        new Money(event.amount()), event.paidOn());
}                                                           // the inbox entry and the change
commit together

// in Claim.recordPayment (rule 12)
if (lastPayment != null && !paidOn.isAfter(lastPayment.paidOn())) return;
```

The listener only translates the message and calls an application service. A message that fails repeatedly goes to a dead-letter queue, where someone finds the cause before it is re-sent. No message broker delivers exactly once and in order.

Spring: `@Transactional` makes the inbox entry and the claim change one commit. The inbox's `@Id` is the event ID: if two copies run at once, the second insert fails and aborts its whole transaction (any write error inside a MongoDB transaction does), the message comes back, and `seen` skips it. That is why the check is a find, not a caught `DuplicateKeyException`.

22. Change several microservices as a chain of steps joined by events, with a named step for failure

Closing a claim must also stop its payments, which live in the payment microservice.

Bad — one transaction across two microservices:

```
@Transactional
public void close(ClaimId id, long version) {
    ...
    claims.save(claim);
    paymentClient.stopPayments(id);                         // another microservice, inside our
transaction
}
```

A transaction cannot cover two databases. If `stopPayments` succeeds and our commit fails, payments are stopped for an open claim.

Good — each step changes one microservice and sends an event that starts the next:

```
1. Claim microservice:   Claim.close  → saves → sends ClaimClosedV1
2. Payment microservice: on ClaimClosedV1 → stops payments → sends PaymentsStoppedV1
3. Claim microservice:   on PaymentsStoppedV1 → Claim.markPaymentsStopped
   If step 2 fails for good: payment microservice sends PaymentsStopFailedV1,
   and we create a task for a caseworker.
```

A failed step cannot be rolled back, so the design names what happens instead, and that is a new business action. Each such process needs design group approval and an architecture decision record (ADR). Do not use events to keep a rule true across two aggregates; that is a sign the two should be one (rule 7).

Part D. Process

23. Get the design note approved before you code a feature with business rules

You are about to build the notes feature.

Bad: start coding, and discover in review that notes should have been inside the claim, or the other way round. Moving children across an aggregate boundary means a data migration.

Good: a one-page design note, approved by the design group first:

```
Purpose:     caseworkers record notes against a claim; a note can need action
Aggregates:  Note (collection "notes", holds claimId); no maximum
States:      OPEN -> ACTIONED (markActioned)
Rules:       a claim cannot close while any note needs action
             enforced in Claim.close, with the count passed in
             example: claim C-1 has 2 notes needing action -> close refused
Events:      NoteNeedsActionV1, NoteActionedV1; used by the claim feature
Indexes:     notes.claimId; notes.(claimId, needsAction)
Talks to:    nothing outside the microservice
Exceptions:  none
Open questions: can a closed claim get a note?
```

The design group decides what each aggregate contains, the rules with worked examples, and any transaction that saves two aggregates. Engineers decide endpoints, classes and tests. Which microservices exist is an ADR. A wrong aggregate boundary is the most expensive mistake to fix later.

24. Mark a feature Simple (data with no rules) or Rich (states and rules) in its design note

You are adding a list of claim types that the business maintains.

Bad: a `ClaimType` aggregate with version checks, events and an outbox, for a list of 12 codes that changes twice a year.

Good: mark it **Simple** in the design note. A Simple feature is data with no business rules: lookup lists, user preferences. Its entity has fields and getters, and checks may sit in the application service.

```
@Document("claim_types")
public record ClaimType(@Id ClaimTypeCode code, String label, boolean active) {}

public interface ClaimTypeRepository extends MongoRepository<ClaimType, ClaimTypeCode> {}
```

Claims, payments and awards are **Rich**: states and rules. Everything in this guide applies to them. The moment a Simple feature gains a rule ("a claim type cannot be deactivated while claims use it"), it becomes Rich and gets an entity method for that rule.

Spring: a record with `@Document` and `@Id` is enough for a lookup list; Spring Data uses the canonical constructor.

25. Fix a mistake with a safe step today and the proper fix by a date

You find that `ClaimService` holds an `if` on status that should be in `Claim`.

Do the safe step now. Tell the design group the same day. Create a ticket with a date. Then do the proper fix and add a test, so the same mistake fails the build next time. Rules 14 and 20 are what make this possible: a feature can change inside without the others noticing.

You find	Today	Proper fix	Never
Two aggregates share a rule that must always be true	Save both in one transaction; add the exception to the design note	Merge them; a migration moves the children inside the parent	Keep the rule true with events
An aggregate gives frequent conflicts or keeps growing	Keep the version check; make the screen say "reload"	Move the part that changes often into its own collection	Turn off the version check
A rule sits in a service, controller or listener	Ticket for this sprint	Move it into the entity method; add a test	Copy the `if` elsewhere
A feature uses another feature's repository	Record today's violations so only new ones fail the build	Add a method on the other feature's application service	Turn the build check off
A change sent no event	Add the missing event to the entity method	A one-off job sends the missed events, agreed with receivers	Let other teams read our database
Messages are in the dead-letter queue	Find the cause with the trace ID and fix it	Re-send them; the inbox skips duplicates	Delete them before you understand them
Our data and another microservice's disagree	Use both audit logs to find which side is right	A job that reports differences; fix through entity methods	A job that silently overwrites one side
Data was repaired by writing to the database	Find the affected records	Repair through entity methods or a reviewed migration	Repair by hand again

26. Keep personal data out of logs, error messages and events

A claim is refused because the National Insurance number is wrong.

Bad:

```
throw new IllegalArgumentException("Invalid NINO: " + nino);     // lands in the logs
public record ClaimOpenedV1(String eventId, String claimId, String nino, String fullName) {}
```

Logs and events are copied to monitoring tools and other teams' systems, which the data protection rules for our database do not cover.

Good:

```
public record Nino(String value) {
    @Override public String toString() { return "Nino(****" + value.substring(5) + ")"; }
}
throw new InvalidNino(claimantId);                          // the ID, not the number
public record ClaimOpenedV1(String eventId, String claimId, String claimantId, LocalDate
openedOn) {}
```

Value objects hide personal data when printed. Events carry IDs, not personal details. If personal data reached production logs, report it through the data protection process.

Part E. Situations you will meet

27. When one button must save two aggregates, first ask whether they are one; then react to an event; only then save both in one approved transaction

The Check answers page of Create claim creates the claim with its first note, and every new claim needs a review task 14 days out.

Bad — two saves in a service method, with no decision made:

```
public ClaimId open(ClaimantId claimantId, ClaimType type, NoteText firstNote) {
    Claim claim = Claim.open(claimantId, type, eligibility, LocalDate.now(clock));
    claims.save(claim);                                    // write 1, atomic on its own
    Note note = Note.create(claim.id(), firstNote, LocalDate.now(clock));
    noteRepository.save(note);                             // write 2; the process dies here:
claim, no note
    return claim.id();
}
```

Half the user's work is saved and the screen shows an error.

Good — three answers, in this order. First, is the second thing really a child? If yes, it goes inside the document (rule 7) and there is one write. Second, can the second thing follow a moment later? The review task can: the claim's own event goes through the outbox, and a listener in our own microservice receives it from the broker like any other message (rule 21) and creates the task:

```
// in Claim.open
events.add(new ClaimOpened(id, today));                    // stored in the outbox with the
claim (rule 19)

// our own listener, fed by the broker: its own transaction, after the claim's commit
@Transactional
public void on(ClaimOpenedV1 event) {
    if (inbox.seen(event.eventId())) return;
    inbox.record(event.eventId());
    reviewTaskService.open(new ClaimId(event.claimId()), event.openedOn().plusDays(14));   // the
other feature's front door (rule 14)
}
```

Third, when the user must see both at once and neither answer fits, save both in one DocumentDB transaction, with the exception written in the design note. The first note is this case:

```
// approved exception "create claim with first note" in the design note
public ClaimId open(ClaimantId claimantId, ClaimType type, NoteText firstNote) {
    Eligibility eligibility = eligibilityClient.check(claimantId);        // before the
transaction (rule 17)
    return tx.execute(status -> {
        Claim claim = Claim.open(claimantId, type, eligibility, LocalDate.now(clock));
        claims.save(claim);
```

```
        noteRepository.save(Note.create(claim.id(), firstNote, LocalDate.now(clock)));   // both
collections, one commit
        audit.record("claim opened with first note", claim.id());
        return claim.id();
    });
}
```

Any of the three answers prevents the half-saved claim; the order is our choice. The boundary question comes first because it is the only answer with no delay and no exception. The event is the normal answer. The transaction comes last because each one is an exception to rule 9 that the design note must carry; it is allowed only because both collections are in our database, and never across microservices (rule 22).

Spring: the listener is a message listener like rule 21's. Not `@TransactionalEventListener(AFTER_COMMIT)`: a crash between the commit and such a listener loses the event, and the outbox never does. DocumentDB runs a multi-document transaction on engine 4.0 and later, through the `MongoTransactionManager` bean.

28. Keep a rule that spans all records of a kind true with a unique database index, and turn the index error into a business error

The rule: a claimant has at most one open claim.

Bad — check, then save:

```
if (claims.existsByClaimantIdAndStatus(claimantId, OPEN)) throw new
ClaimantAlreadyHasOpenClaim(claimantId);
claims.save(Claim.open(claimantId, ...));                    // two caseworkers do this at the
same second
```

Both checks pass, both saves succeed, and the claimant has two open claims.

Good — the database holds the rule; the service translates its error:

```
@Document("claims")
@CompoundIndex(name = "one_open_claim_per_claimant", def = "{'claimantId': 1}", unique = true,
               partialFilter = "{'status': 'OPEN'}")
public class Claim { ... }
```

```
// in ApiErrors, the walkthrough's error handler
@ExceptionHandler(DuplicateKeyException.class)
ProblemDetail duplicate(DuplicateKeyException e) {
    if (e.getMessage().contains("one_open_claim_per_claimant"))
        return problem(HttpStatus.UNPROCESSABLE_CONTENT, "Business rule violated",
"CLAIMANT_HAS_OPEN_CLAIM", "...");
    return bug(e);                                           // any other unique index is a bug:
500 (API guide, rule 26)
}
```

An aggregate guarantees only rules about itself; a rule about all claims is the database's, and the index is the one check two users cannot race past. Keep the `exists` check for a friendly error in the common case. The error is translated at the edge, because a write error aborts the DocumentDB transaction and the service has nothing left to do.

Spring: `@CompoundIndex` with `partialFilter` describes a partial unique index, which DocumentDB supports on engine 5.0 and later; the migration creates it (automatic index creation stays off). A violation surfaces as Spring's `DuplicateKeyException`, which the shared `@RestControllerAdvice` maps.

29. Create an entity through a named factory method that checks the creation rules and raises the event; keep the constructor private

You are writing the code that creates a claim.

Bad:

```
Claim claim = new Claim();
claim.setClaimantId(claimantId);
claim.setStatus(OPEN);
claims.save(claim);
```

Nothing checks eligibility or the claim type, and no `ClaimOpened` event is sent.

Good:

```
@Document("claims")
public class Claim {
    @PersistenceCreator
    private Claim(ClaimId id, ClaimantId claimantId, ClaimTypeCode type, LocalDate openedOn) {
... }

    public static Claim open(ClaimantId claimantId, ClaimType type, Eligibility eligibility,
LocalDate today) {
        if (eligibility != ELIGIBLE) throw new ClaimantNotEligible(claimantId);
        if (!type.active()) throw new ClaimTypeNotActive(type.code());
        Claim claim = new Claim(ClaimId.next(), claimantId, type.code(), today);   // the type's
code, not the type (rule 8)
        claim.events.add(new ClaimOpened(claim.id, today));
        return claim;
    }
}
```

Creation is a business action like any other, so it has a business name and its rules live in one place. The only `new Claim(...)` is inside `Claim` and inside Spring Data when it loads a document. We make our own IDs (`ClaimId.next()`), so the document's `_id` is never left to the database.

Spring: `@PersistenceCreator` marks the constructor Spring Data uses when reading a document, so it can stay private and skip the creation rules; the public factory keeps them.

30. Read for a screen through a query class that reads the collections directly; never build a screen out of aggregates

The claim list shows the claim ID, status, date opened, number of open notes and last payment date, for 50 claims a page.

Bad — load aggregates and pick fields out:

```
List<ClaimRow> rows = new ArrayList<>();
for (Claim claim : claims.findByStatus(OPEN, page)) {         // 50 full documents with evidence
    int open = noteService.countNeedingAction(claim.id());   // 50 more queries
    rows.add(new ClaimRow(claim.id(), claim.status(), open, claim.lastPayment()));
}
```

Fifty-one database round trips for one page, and the aggregate code now has getters that exist only for screens.

Good — a query class, two queries, a flat record:

```
@Component
public class ClaimListQuery {
    public List<ClaimRow> openClaims(int page) {
        Query q = query(where("status").is("OPEN")).with(Sort.by(DESC, "openedOn")).skip(page *
50L).limit(50);
        q.fields().include("status", "openedOn", "lastPayment.paidOn");
        List<ClaimRow> rows = mongo.find(q, ClaimRow.class, "claims");          // query 1: the
page

        Map<String, Integer> openNotes = mongo.aggregate(newAggregation(          // query 2:
counts for those 50
                match(where("claimId").in(ids(rows)).and("needsAction").is(true)),
                group("claimId").count().as("open")),
            "notes", OpenNoteCount.class).getMappedResults().stream()
            .collect(toMap(OpenNoteCount::claimId, OpenNoteCount::open));

        return rows.stream().map(r -> r.withOpenNotes(openNotes.getOrDefault(r.claimId(),
0))).toList();
    }
}

public record ClaimRow(@Id String claimId, String status, LocalDate openedOn, Payment
lastPayment, int openNotes) {   // @Id: filled from _id
    public record Payment(LocalDate paidOn) {}
}
public record OpenNoteCount(@Id String claimId, int open) {}      // $group puts its key in _id
```

Aggregates and repositories exist to change data safely. Reads have no rules to protect, so they go straight to the collections, read across aggregates freely, and return exactly what the screen needs. A query class never changes anything, and a rule never lives in a query.

Spring: `MongoTemplate.find` with `fields().include(...)` reads only the fields the row needs into a record that is not the entity; `mongo.aggregate` runs a `$match` and `$group` pipeline. A list query may use `secondaryPreferred` reads; an application service never does, because DocumentDB replicas are eventually consistent.

31. Search and reports are queries too; give a slow one its own read collection filled by events

The business wants a report of claims closed per month with total payments.

Bad — compute it from aggregates at request time:

```
for (Claim claim : claims.findAll()) { ... }                // 2 million documents
```

The report reads two million documents on every request and takes minutes.

Good — a query over the collections (rule 30). When that is too slow, keep a separate read collection that the domain events fill:

```
@EventListener
public void on(ClaimClosed e) {
    mongo.upsert(query(where("month").is(YearMonth.from(e.closedOn()).toString())),
                 new Update().inc("closedCount", 1), "monthly_closure_report");
}
```

```
mongo.find(new Query().with(Sort.by("month")), MonthlyClosure.class, "monthly_closure_report");
```

The read collection is the second step, taken only when the query has been measured too slow, because it is one more thing to keep right. It is a copy, built from events, and can be rebuilt by replaying the outbox. Nothing is ever changed through it.

Spring: `MongoTemplate.upsert` with `Update.inc` adds to the month's counter in one atomic write.

32. Do not store on a parent what a child aggregate knows; ask for it when reading, or copy it through an event and accept a short delay

The claim page shows "has notes needing action".

Bad — a flag on the claim, set by the note feature:

```
// in NoteService.markNeedsAction
Claim claim = claims.findById(note.claimId()).orElseThrow();
claim.setHasOpenNotes(true);                                // another feature changes our
aggregate
claims.save(claim);
```

Two aggregates are saved in one action, the flag drifts when a note is actioned on another path, and the note feature now depends on the claim's internals.

Good — the normal answer: ask when reading (rule 30, query 2). When the flag must be on the claim itself, for a rule or for speed, copy it through an event:

```
// our own listener for NoteNeedsActionV1, from the broker, with the inbox (rules 21 and 27)
@Transactional
public void on(NoteNeedsActionV1 event) {
    if (inbox.seen(event.eventId())) return;
    inbox.record(event.eventId());
    ClaimId claimId = new ClaimId(event.claimId());
    Claim claim = claims.findById(claimId).orElseThrow(() -> new ClaimNotFound(claimId));
    claim.noteNeedsAction(new NoteId(event.noteId()));        // a method, not a setter; records
```

```
the note ID
    claims.save(claim);
}
```

The copy is a moment behind, and it bumps the claim's version, so a caseworker with the claim open must reload; the design note says both. For the real rule, `close` still asks for the count at the moment of closing (rule 1), because a copy is never the truth.

33. Never delete a business record; add a business action that marks it, with a reason

A caseworker created a claim for the wrong claimant.

Bad:

```
claims.deleteById(claimId);
```

The audit trail, the notes that point at the claim and the events already sent now point at nothing, and the payment microservice may still act on it.

Good:

```
public void cancel(CancellationReason reason, LocalDate today) {
    if (status == CLOSED) throw new ClaimAlreadyClosed(id);
    status = CANCELLED;
    cancelledOn = today;
    events.add(new ClaimCancelled(id, reason, today));
}
```

Lists hide cancelled claims by default. A deletion required by the data protection rules is a separate, approved process that removes personal data and keeps the document's skeleton. A build check fails on any call to `deleteById`, `delete` or `remove` outside that process.

34. Run a rule that depends on time from a scheduled job that calls the same entity method, one record per transaction

A claim with no activity for 90 days is closed automatically.

Bad — a job that updates the collection:

```
mongo.updateMulti(query(where("lastActivityOn").lt(cutoff)), update("status", "CLOSED"),
"claims");
```

No rule check, no event, no audit, no version bump; the payment microservice never hears about it.

Good:

```
@Scheduled(cron = "0 0 2 * * *")                              // 02:00 every day
public void closeInactiveClaims() {
    LocalDate today = LocalDate.now(clock);
    for (ClaimId id : claims.findIdsInactiveSince(today.minusDays(90))) {
        tx.executeWithoutResult(status -> closeOne(id, today));   // one transaction each (rule
```

```
9)
    }
}

private void closeOne(ClaimId id, LocalDate today) {
    Claim claim = claims.findById(id).orElseThrow(() -> new ClaimNotFound(id));
    int open = noteService.countNeedingAction(id);
    claim.closeForInactivity(open, today);                  // the entity decides; refuses if
notes are open
    claims.save(claim);
    audit.record("claim closed for inactivity", id);
}
```

The job is just another caller of the entity, like the controller and the message listener. The time comes from the `Clock` bean, so a test can set today to any date.

Spring: `@Scheduled(cron = ...)` with `@EnableScheduling`. On more than one instance, the job first takes a lock document (`findAndModify` on a `job_locks` collection, with an expiry); an instance that finds it taken skips the run.

35. When a process has its own steps and deadlines, give it its own aggregate

Closing a claim with payments takes three steps across two microservices (rule 22), and the business wants to see which step each claim is at, and chase one that is stuck for two days.

Bad — flags on the claim:

```
private boolean closeRequested, paymentsStopped, paymentsStopFailed, letterSent;
```

The claim's methods fill up with process `if`s, and nobody can see the whole process in one place.

Good — a `ClaimClosure` aggregate, its own collection, that holds the state of the process and reacts to the events:

```
@Document("claim_closures")
public class ClaimClosure {
    @Id private ClaimId claimId;                            // one closure per claim
    @Version private long version;
    private ClosureStep step;                               // REQUESTED, PAYMENTS_STOPPED,
LETTER_SENT, DONE, STUCK
    @Field(targetType = FieldType.STRING) private LocalDate stepStartedOn;   // a UK date (rule
13)
    @Transient private final List<DomainEvent> events = new ArrayList<>();   // and
@DomainEvents, as in Claim

    public void paymentsStopped(LocalDate today) {
        if (step != REQUESTED) return;                      // late or repeated event: ignore
(rule 21)
        step = PAYMENTS_STOPPED;
        stepStartedOn = today;
        events.add(new ClosureLetterRequested(claimId));
    }

    public void checkDeadline(LocalDate today) {            // called by a daily job (rule 34)
        if (step == DONE || step == STUCK) return;
```

```
        if (today.isAfter(stepStartedOn.plusDays(2))) {
            events.add(new ClosureStuck(claimId, step));      // the step it was stuck at
            step = STUCK;
        }
    }
}
```

The claim keeps its own rules; the process keeps its steps. This is the "directing microservice" of the API guide, at the scale of one aggregate.

36. Change an aggregate's shape behind a flag, with a migration that uses the entity's own methods

Notes were inside the claim; the design group decided they are their own aggregate (rule 7).

Bad — a script on release night:

```
db.claims.find().forEach(c => c.notes.forEach(n => db.notes.insertOne({ ...n, claimId: c._id
})));
```

Half the claims are migrated when the script fails, the old code reads the rest, and the two sets drift.

Good:

1. Release 1: the new `Note` document and repository exist; new notes go to `notes`; reads check both places behind a feature flag.
2. A migration job, one claim per transaction (rule 9), creates the `Note` aggregates through `Note.create` and marks the claim as migrated; it can be rerun.
3. Release 2: the flag is on for everyone; the old array is read by nobody.
4. Release 3: a final job unsets the old array (`Update.unset("notes")`), and the flag is removed.

The migration goes through the entity's methods so that every rule and event still holds. Rule 25's "proper fix" column is this list.

Spring: a field that no longer exists in the class is simply ignored when a document is read, so release 2 can ship before the array is removed. Create any new index in the migration, not with `auto-index-creation`.

37. Test an entity with no database, a service with a fake repository, and the repository alone against a real database

You are writing the tests for "close claim".

Bad — every test starts the whole microservice and a database:

```
@SpringBootTest
void closeRefusedWhenNotesOpen() {
    mongo.insert(claimDocument); mongo.insert(noteDocument);  // 40 lines of setup
    mvc.perform(post("/v1/claims/9f1c/close")...);
    ... status().isUnprocessableContent()
}
```

Ten seconds per test, and the rule being tested is buried under HTTP and documents.

Good — three layers, each tested for what it owns:

```
// the entity: the rule, in memory, in milliseconds; plain JUnit, no Spring
@Test void closeRefusedWhenNotesOpen() {
    Claim claim = Claim.open(claimantId, ClaimTypes.standard(), ELIGIBLE, date("2026-10-01"));
    assertThrows(ClaimHasOpenActions.class, () -> claim.close(2, date("2026-10-31")));
}

// the service: the steps, with a hand-written fake repository, a mock of the other feature and a
fixed Clock; no Spring
@Test void closeSavesAndAudits() {
    ClaimService service = new ClaimService(fakeClaims, mock(NoteService.class), fakeAudit,
Clock.fixed(...));
    service.close(claimId, 7);
    assertTrue(fakeClaims.saved(claimId));
    assertEquals("claim closed", fakeAudit.last());
}

// the repository: the mapping, against a real MongoDB in a container; one test class
@DataMongoTest @Testcontainers
class ClaimRepositoryTest {
    @Container @ServiceConnection static MongoDBContainer mongo = new
MongoDBContainer("mongo:8.0").withReplicaSet();
    @Test void saveAndLoadRoundTrip() { ... }
}

// the adapter: the translation, against WireMock playing the other microservice; one test class
per adapter
@WireMockTest
class EligibilityClientTest {
    @Test void translatesTheirAnswer(WireMockRuntimeInfo wm) {
        stubFor(get("/v1/eligibility/c88").willReturn(okJson("{\"status\":\"NOT_ELIGIBLE\"}")));
        assertEquals(Eligibility.NOT_ELIGIBLE, client(wm).check(new ClaimantId("c88")));   //
client(wm): the adapter over WireMock's URL
    }
    @Test void timeoutBecomesOurError(WireMockRuntimeInfo wm) {
        stubFor(get("/v1/eligibility/c88").willReturn(ok().withFixedDelay(3000)));
        assertThrows(EligibilityUnavailable.class, () -> client(wm).check(new
ClaimantId("c88")));
    }
}
```

Most tests are entity tests, because that is where the rules are. One `@WebMvcTest` per endpoint proves the permission check (rule 15). The adapter tests are the only place the other system's JSON appears. The outbox sender, each message listener and the object-store adapter are tested against Floci, which plays the AWS services locally (like LocalStack): the test publishes a `PaymentMadeV1` to the local queue and asserts on the claim. DocumentDB has no local edition, so the repository test runs against MongoDB; a smoke test against a real cluster runs in the pipeline.

Spring: `@DataMongoTest` loads only the Mongo slice; `@ServiceConnection` on the Testcontainers container sets the connection for it; `withReplicaSet()` is needed for transactions. `@WireMockTest` starts a WireMock server for the adapter test and hands its URL to the test method. Mockito's `mock` comes with `spring-boot-starter-test`.

38. Show two aggregates on one screen and save them with two buttons; one button that saves two things is a design smell to raise

The claim page shows the claim and lets the caseworker edit contact details and add a note.

Bad — one Save button that sends both:

```
@Transactional
public void saveClaimPage(ClaimId id, long version, ContactDetails contact, NoteText noteText) {
    Claim claim = claims.findById(id).orElseThrow(() -> new ClaimNotFound(id));
    claim.requireVersion(version);
    claim.changeContact(contact);
    claims.save(claim);
    noteRepository.save(Note.create(id, noteText, LocalDate.now(clock)));    // two aggregates,
one button (rule 27)
}
```

One button is one request that saves two aggregates, so every claim page becomes an exception to rule 9 that nobody approved, and the note has no version of its own.

Good — each part of the screen saves its own aggregate with its own button and its own version:

```
[Contact details]   Save contact details  -> ClaimService.changeContact(claimId, version,
contact)
[Add a note]        Add note              -> NoteService.add(claimId, noteText)
```

When a screen keeps needing one button for two aggregates, say so to the design group: either the boundary is wrong (rule 7) or the action deserves a name of its own (rule 27).

Who to ask

Ask a senior engineer the same day when you do not know where code belongs. Ask the design group, before you code, for a new or changed aggregate, a two-aggregate transaction or a new link between microservices. Tell the receiving teams before you change a published event. Report a security or personal data problem through the incident process at once.

Still to fill in: design group members, team channel, design session time, incident process link.