# Jab Table Header Badal Jata Hai

Ek layout bug kabhi sirf ek chhota visual glitch nahi hota. Jo screenshot mein dikha tha, usmein source table ka header same tha, lekin page 4 aur page 5 par header ka rendered form alag dikh raha tha. Page 4 par kuch labels chhote, clipped, ya incomplete lag rahe the. Page 5 par wahi repeated header zyada complete wrap ke saath aa raha tha.

Iska matlab yeh nahi ki markdown badal gaya. Iska matlab hai ki renderer ne same semantic object ko do alag raaston se process kiya. Pehla raasta tha original table fragment ka header. Doosra raasta tha continuation page par repeat kiya gaya header. Jab in dono raaston ki measurement, wrapping, row height, clipping, padding, ya baseline policy ek jaisi nahi hoti, tab same content visually different ban jata hai.

Yeh book us problem ko alag alag abstraction levels par naam deti hai. Naam important hai, kyunki jab problem ka naam clear hota hai, debugging bhi clear hoti hai. Agar hum sirf bolte rahein “header weird hai”, toh fix random patch ban sakta hai. Agar hum bolte hain “fragment-stable header invariant violate hua hai”, toh system ko pata hota hai kya trace karna hai, kya assert karna hai, aur kya audit banana hai.

# Chapter 1: Visual Symptom

Sabse neeche ka level hai visual symptom. User ke liye bug wahi hota hai jo aankh ko dikhta hai. Is case mein symptom tha: consecutive pages par table header text different dikh raha tha.

Page 4 par table header mein kuch columns ka text truncated feel ho raha tha. For example, “Publicly evidenced realised cycle” jaise full label ke bajay sirf “Publicly evidenced” jaisa short visual result dikh raha tha. Dusre page par same header zyada complete form mein wrap ho raha tha: “Publicly evidenced realised cycle”, “Publicly evidenced rejection / hold pattern”, “Escalation route visible in public sources”, and so on.

Visual symptom ko formal naam diya ja sakta hai:

- inconsistent repeated table header
- header text clipping
- header truncation
- continuation header mismatch
- non-identical table header rendering

Is level par hum technical root cause nahi bol rahe. Hum sirf artifact describe kar rahe hain. Yeh important hai, kyunki visual bug report ko neutral hona chahiye. Agar user screenshot bhejta hai aur hum turant bol dein “yeh row-height bug hai”,

toh shayad hum galat direction mein chale jayenge. Visual report ka kaam hai observation ko preserve karna:

“Same table ka header page 4 aur page 5 par same nahi dikhta. Source content stable hai, par rendered text aur wrapping different hai.”

Good visual diagnosis mein teen cheezein honi chahiye:

1. Kya dikh raha hai.

2. Kahan dikh raha hai.

3. Kya expected tha.

Yahan:

Kya dikh raha hai: repeated header text changes across fragments.

Kahan dikh raha hai: split table ke first fragment aur continuation fragment mein.

Expected kya tha: same semantic header should produce same resolved header layout on every fragment, unless author ne continuation-specific abbreviated header explicitly configure kiya ho.

Visual symptom ke level par ek aur subtle point hai. Kabhi kabhi text actually missing nahi hota, balki clip box ke bahar paint hota hai. Kabhi kabhi text paint hota hai par background ya overprint usse hide kar deta hai. Kabhi line wrap hui hoti hai, par row height kam hone ke karan second line invisible ho jati hai. Isliye visual symptom ko “data lost” nahi bolna

chahiye jab tak trace prove na kare. Better phrase hai: “rendered header is incomplete or clipped.”

Iska debugging implication simple hai: screenshot se hum detection start karte hain, conclusion nahi. Pehla detector page images compare karega. Dusra detector PDF text extraction karega. Teesra detector render trace dekhega. Agar source text present hai aur page 5 par dikhta hai, par page 4 par nahi, toh bug likely source parser mein nahi, layout fragmentation ya paint clipping mein hai.

Visual symptom ka goal hai human observation ko machine-checkable incident mein convert karna.

# Chapter 2: Typography And Paged Media

Typography ke perspective se yeh problem “table header repetition failure” hai. Paged media systems mein tables often page break ke across split hote hain. Jab table agle page par continue hota hai, reader ko context dene ke liye header repeat hota hai. Is repeated header ko running table header, continued table head, ya repeated table head bola ja sakta hai.

Ek achhi book, report, ya academic PDF mein split table ke continuation page par header repeat hona normal hai. Lekin repeated header ka rule simple nahi hai. Header repeat karna kaafi nahi; header must preserve table semantics and reader context.

Paged media level par expected behavior:

- first page header and continuation header semantically same ho
- column count same ho
- column widths same ho
- header row height enough ho for wrapped labels
- header background, rules, padding, and typography consistent ho
- continuation marker optional ho, but explicit ho

Problem tab hoti hai jab first fragment ka header aur repeated fragment ka header alag typographic treatment lete hain. For example, first page ka header one-line height reserve karta hai, lekin repeated page ka header multi-line height reserve karta hai. Reader ko lag sakta hai ki second page par table ka schema change ho gaya. Agar schema change ka illusion aa gaya, toh document trust lose karta hai.

Formal typography names:

- broken running table header
- repeated table head mismatch
- table continuation header inconsistency
- unstable header row across page breaks
- continuation context failure

Yeh problem sirf aesthetics nahi hai. Tables are cognitive maps. Header labels tell reader how to interpret each column. Agar label clipped hai, toh data ka meaning ambiguous ho jata hai. Financial, medical, legal, policy, or research reports mein yeh serious issue hai.

Is level par ek useful analogy hai. Running header in a book usually chapter title ya section title repeat karta hai. Agar left page par “Chapter 3” aur right page par galat chapter title aaye, reader confused hota hai. Table header bhi mini-running-header hai, but table ke local scope mein. Isliye repeated table header must be stable.

Typography systems traditionally isko solved problem maante hain, but implementation hard hai because header height depends on column width, font, language, hyphenation, bold weight, padding, and available page region. Agar table narrow hai, header wraps more. Agar font changes, line height changes. Agar target page size A4 se trade paperback ho jaye, same header ka resolved height different ho sakta hai. Lekin once final table grid resolve ho gaya, us grid ke andar header layout stable rehna chahiye across fragments.

Important distinction:

Table header ka final layout target-specific ho sakta hai. A4 aur trade paperback mein header different ho sakta hai. That is fine.

Same target ke same table ke fragments mein header different nahi hona chahiye. That is the bug.

Paged media contract bolta hai: table split ho sakta hai, par table ka schema split nahi hona chahiye.

# Chapter 3: Layout Engine

Layout engine ke level par yeh problem “fragmentation non-idempotence” hai.

Idempotence ka simple meaning: same input ko same context mein process karo, result stable rahe. Table header case mein input hai semantic header row plus final column grid. Context hai same font, same preset, same page target, same table width. Agar renderer first fragment aur continuation fragment ke liye header ko separately recompose karta hai aur alag output deta hai, toh operation idempotent nahi hai.

Fragmentation ka meaning: ek block ko multiple page fragments mein todna. Table ek semantic object hai. Jab table page boundary cross karta hai, renderer uske fragments banata hai:

- fragment 0: first part
- fragment 1: continuation
- fragment 2: further continuation

Problem tab aati hai jab fragment 0 aur fragment 1 independently header layout decide karte hain. Fragment 0 bolta hai: header row height 42pt enough hai. Fragment 1 bolta hai: header row height 78pt chahiye. Dono same source se aaye, phir bhi geometry different.

Formal layout-engine names:

- fragmentation non-idempotence
- continuation-fragment reflow divergence

- split-table header measurement drift
- geometry/content disagreement
- fragment-local recomposition bug
- first-fragment vs continuation-fragment policy split

Is type ke bug ka root cause aksar yeh hota hai:

```
measure_original_header(table)
measure_repeated_header(table)
```

Yeh dono functions similar lagte hain, but exact same nahi hote. Ek padding include karta hai, dusra nahi. Ek bold metrics use karta hai, dusra body metrics. Ek wraps after slash, dusra slash ko unbreakable treat karta hai. Ek clip rectangle row height ke equal set karta hai, dusra natural height ke equal. Bas phir visual divergence aa jata hai.

Correct model yeh hona chahiye:

```
canonical_header = resolve_header_layout(table,
 final_column_grid)

for fragment in table_fragments:
    fragment.header = replay(canonical_header)
```

Yahan “replay” ka matlab blind copy nahi. Page y-coordinate different ho sakta hai. Fragment phase different ho sakta hai. But header ke internal facts stable hone chahiye:

- cell text sequence
- cell width
- inner padding
- text wraps
- row height
- baseline positions relative to header box
- clip bounds
- border/rule policy

Layout engine level par trace extremely important hai. Har table ke liye renderer ko log karna chahiye:

- table id
- target page size
- final outer width
- column widths

- header natural height
- header reserved height
- header painted clip height
- fragment id
- fragment phase
- whether header was canonical replay or recomposed

Tab bug instantly visible ho jayega:

```
1 table=node-89 header natural=78pt reserved=
  42pt fragment=0
2 table=node-89 header natural=78pt reserved=
  78pt fragment=1
```

This is the smoking gun. It tells us text source fine hai, width fine hai, but first fragment reserved wrong height.

Layout engine mein ek golden rule hai: measurement and painting must share the same geometry. Agar measurement 42pt bolti hai aur paint 78pt worth text draw karta hai, overlap hoga. Agar measurement 78pt bolti hai par paint clip 42pt ka use karta hai, truncation hoga. Agar measurement and fragmentation 78pt use karte hain but continuation header 42pt use karta hai, page fill unstable hoga.

Isliye formal root issue ho sakta hai: “measurement-paint geometry divergence.”

# Chapter 4: Data Model

Data model ke level par yeh problem “duplicated derived state” hai.

Source markdown mein header ek hi jagah tha. But renderer pipeline mein woh information multiple forms mein convert hoti hai:

- source AST
- semantic table node
- publication table row
- table cell components
- layout measurement records
- fragmentation records
- placement records
- paint commands
- PDF text/glyph output

Yeh transformation chain normal hai. Problem tab hoti hai jab derived state duplicate ho kar independent ho jata hai. For example, semantic header row ek jagah hai, repeated header ke liye copied metadata doosri jagah hai, aur paint code teesri jagah se labels read karta hai. Agar in derived copies ka lifecycle controlled nahi hai, toh stale ya inconsistent output possible hai.

Formal data/model names:

- duplicated derived state
- stale derived view
- multi-source-of-truth layout state
- derived representation drift
- denormalized layout metadata mismatch
- semantic identity loss

Single source of truth ka matlab yeh nahi ki renderer mein sirf ek object hoga. Rendering pipelines naturally derived data banati hain. Single source of truth ka real matlab hai: har derived object ka authority clear ho.

Example:

- semantic content authority: source/semantic IR

- column grid authority: table layout solver
- header geometry authority: canonical header layout record
- fragment placement authority: fragmentation solver
- paint authority: placement display list

Agar paint code directly semantic header se text read karta hai but geometry placement record se leta hai, toh dangerous hai. Kyunki content and geometry alag authorities se aa rahe hain. Better hai paint code ek resolved display list consume kare jisme text, x, y, width, height, clip, font, and source identity already consistent hon.

Yeh identity-preserving renderer ke liye central point hai. Har mark traceable hona chahiye, but traceability ka matlab sirf

“yeh glyph source line 25 se aaya” nahi. Traceability should also answer:

- Is glyph ka layout decision kis solver ne liya?
- Iske column width ka origin kya tha?
- Is header row height ka authority kaun tha?
- Is fragment par repeated header canonical replay hai ya recomposed?
- Agar clip hua, clip rectangle kis node ne derive kiya?

Data model bug ka symptom page par dikhta hai, but cause often metadata lifecycle mein hota hai. Ek stale copy hold kar leti hai old width. Ek late-stage code path font weight override miss kar deta hai. Ek repeated header node semantic role preserve karta hai but layout role nahi. Yeh sab “data drift” ke forms hain.

Correct system mein table header ke liye ek canonical record ho sakta hai:

```
ResolvedTableHeader {
    table_id,
    column_grid_id,
    cells,
    row_height,
    display_items,
    source_span,
    derivation_trace
}
```

Fragments then refer to this record. They may translate it vertically, but should not reinterpret it.

Data model ka moral simple hai: semantic identity preserve karo, but derived geometry ko bhi identity do. Sirf content tracing enough nahi; decision tracing bhi chahiye.

# Chapter 5: Software Design And Connascence

Software design ke level par yeh connascence problem hai.

Connascence ka matlab hai coupling ka woh form jahan ek part change ho toh doosra part bhi change karna padta hai. Har coupling bad nahi hoti. Problem tab hoti hai jab coupling hidden ho. Table header case mein likely multiple modules same rule “know” kar rahe the:

- header measurement kaise hota hai
- header row height kaise choose hoti hai
- continuation header kaise repeat hota hai
- clipping box ka size kya hota hai
- padding kitna hai
- border/rule kahan paint hota hai

Agar yeh rules duplicated hain, toh ek function update karne se dusra stale reh sakta hai. Result: first page aur continuation page disagree karte hain.

Formal software-design names:

- connascence of algorithm
- connascence of execution
- connascence of timing

- connascence of position
- DRY violation in layout policy
- parallel implementation of the same rule
- split authority

Connascence of algorithm: do places same algorithm independently implement karte hain. Example: `measure_header_height` and `paint_repeated_header` dono wrapping logic calculate karte hain. Agar ek hyphenation enable karta hai aur dusra nahi, bug.

Connascence of execution: system assumes measurement paint se pehle run hoga, aur paint measurement result use karega. Agar continuation path measurement bypass karta hai, bug.

Connascence of timing: width resolve hone se pehle header measure ho gaya. Later width changed, but header cache invalidate nahi hua. Bug.

Connascence of position: row array index 0 ko header maana gaya, but split fragment mein row index shifted ho gaya. Page 5

par continuation row ka first body fragment header ke saath visually confuse ho gaya. Bug.

Least connascence ka design yeh bolta hai: rules ko data bana do, and the stages consume the same resolved data. Header wrapping ek function mein ho, header display list ek canonical result ho, paint bas us display list ko paint kare. Fragmentation uski height use kare. Audit usi record ko compare kare.

Bad shape:

```
fragmenter measures header
painter paints header
continuation builder repeats header
auditor guesses header from PDF text
```

Better shape:

```
header resolver creates canonical header layout
fragmenter reserves canonical header height
painter replays canonical header display items
auditor compares emitted header instances
against canonical id
```

Is approach ka benefit hai ki fix local nahi, systemic hota hai. Aaj table header bug fix hua. Kal code block gutter, footnote anchor, table width, and heading keep-with-next bugs bhi same architecture se easier ho jayenge.

Connascence ka practical test:

“Agar main header padding change karun, mujhe kitni files update karni padengi?”

Answer one hona chahiye: style or resolver policy. Agar answer three hai, system fragile hai. Agar answer unknown hai, trace missing hai.

# Chapter 6: Invariant And Contract

Invariant ka matlab hai woh rule jo system ke har valid output mein true hona chahiye. Table header ke liye main invariant hai:

“Given a table id, final column grid, font context, and page target, every repeated header instance must be geometrically and textually equivalent to the canonical header layout, except for explicit continuation markers.”

Hinglish mein simple version:

“Same table ka same header, same page target mein, har fragment par same tarah wrap, reserve, clip, and paint hona chahiye.”

Formal names:

- fragment-stable header contract
- repeatable artifact invariant
- canonical header replay invariant
- measurement-paint consistency contract

- table schema continuity invariant

Is invariant ko precise karna zaroori hai. “Same dikhe” vague hai. Machine ko compare karne ke liye fields chahiye:

- text sequence same
- cell count same
- column x offsets same
- column widths same
- row height same
- cell padding same
- line breaks same
- baseline offsets same
- clip bounds same
- background and rule positions same

Exception allowed ho sakti hai:

- continuation marker add karna, like “continued”

- repeated caption omit karna, if preset says so
- top border style change karna, if border policy says continuation seam

But exceptions explicit policy records mein hone chahiye. Silent divergence not allowed.

Invariant violation ka impact direct hai. Agar header row under-measured hai, table body wrong y start pe aa sakta hai. Agar header over-measured hai, page fill gap badh sakta hai. Agar continuation header taller hai than first header, table fragments page capacity differently consume karte hain. Then footnotes, bottom content, and page breaks bhi affected ho sakte hain.

Contracts should live near renderer boundaries. For example:

Before fragmentation:

```
assert table.column_grid.resolved
assert table.header_layout.resolved
assert header_layout.width == table.outer_width
```

During fragmentation:

```
assert fragment.reserved_header_height ==
table.header_layout.row_height
```

During painting:

```
assert paint_clip.height ==
fragment.reserved_header_height
```

After PDF emission:

```
assert extracted_header_text(fragment_n) ==
canonical_header_text
assert glyph_boxes_within_header_clip
```

Yeh layered contract useful hai because every stage catches a different failure. Compile-time types help with missing data. Runtime assertions catch impossible geometry. Visual audit catches backend and font surprises.

Good invariant design ka rule:

The earlier the failure can be detected, the cheaper it is. But the final PDF still must be audited, because typography is an artifact problem, not only a data problem.

# Chapter 7: Testing And Audit

Testing level par yeh “metamorphic visual regression” hai.

Metamorphic test ka idea simple hai: exact golden output hamesha necessary nahi, but relation between outputs must hold. Yahan relation hai:

“Same table header, same resolved grid, multiple fragments: header instances must match.”

We do not need to know every pixel in advance. We need to know that fragment 0 header and fragment 1 header are equivalent under the allowed continuation policy.

Formal testing names:

- metamorphic property failure
- visual regression
- snapshot instability
- cross-fragment consistency failure
- PDF extraction invariant failure
- layout oracle gap

SOTA audit system ko multiple signals combine karne chahiye:

1. Trace audit

Trace se check karo:

- table id same hai?
- fragment ids kya hain?
- header layout id same hai?

- column grid id same hai?
- reserved height same hai?
- fallback path triggered hua?
- first fragment and continuation fragment same canonical header use kar rahe hain?

2. Geometry audit

PDF ya emitted placement data se check karo:

- header boxes page bounds ke andar hain?
- header text boxes clip box ke andar hain?
- body row header ke neeche start ho raha hai?
- repeated header ka height canonical height ke equal hai?

3. Text audit

Extracted text se check karo:

- source header labels present hain?
- continuation header mein labels missing toh nahi?

- first header aur repeated header equivalent hain?
- body continuation text header row mein accidentally mix toh nahi ho raha?

4. Pixel audit

Page raster image compare karo:

- obvious overlap
- clipped text
- large blank areas
- table overflow
- footnote collision
- heading stranded at footer bottom

Pixel audit alone enough nahi, because it can say “something looks different” but not why. Trace audit alone enough nahi,

because trace can be internally consistent but PDF backend can still clip. Best system dono combine karta hai.

Ek practical detector:

```
for each table_id:
    canonical = trace.table[table_id].header_layout
    for each fragment in table.fragments:
        observed = extract_header_instance(fragment)
        compare_text(canonical, observed)
        compare_geometry(canonical, observed)
        compare_clip(canonical, observed)
```

Finding output should be actionable:

```
table=node-89
fragment=0
problem=header_reserved_height_less_than_natural_height
canonical_header_height=78pt
reserved_header_height=42pt
missing_labels=["realised cycle", "rejection / hold pattern"]
likely_stage=fragmentation/measurement
```

This is how debug infra stops guesswork. Instead of asking “why does page 4 look weird?”, it says “the first fragment clipped canonical header because reserved height was lower than natural header height.”

Testing philosophy:

- Unit tests prove the resolver.
- Integration tests prove fragmentation.

- Visual audits prove artifact.
- Trace audits prove derivation.
- Metamorphic tests prove consistency across transformed contexts.

For a PDF renderer, real confidence tab aata hai jab all five exist.

# Chapter 8: System Shape We Actually Want

Ab final system ka ideal shape.

Renderer ko sirf PDF banana nahi chahiye. Renderer ko explainable PDF banana chahiye. Har visible mark ke peeche derivation chain honi chahiye:

Source -> Semantic -> Publication -> Component -> Layout -> Composition -> Fragment -> Placement -> Paint -> PDF.

But this chain useful tab hai jab decisions bhi trace hon:

- Why this width?
- Why this break?
- Why this row height?

- Why this page?
- Why this footnote position?
- Why this header repeat?
- Why this clip rectangle?

Table header bug se lesson yeh hai: artifact correctness needs decision provenance.

Ek robust architecture:

1. Semantic identity stable rakho.

Every table, row, cell, paragraph, footnote, heading, and code block has stable identity.

2. Derived geometry ko named records do.

Column grid, header layout, row heights, code shell metrics, footnote reservations all get ids.

3. Measurement and painting ko decouple mat karo without a shared contract.

Painter should not re-decide layout. Painter should render resolved placement.

4. Fragmentation ko canonical components replay karne do.

Continuation fragments should not invent new header geometry.

5. Audit findings should point to the derivation break.

Bad finding: “page 4 header bad.”

Good finding: “table node-89 fragment 0 reserved 42pt for a header whose canonical natural height is 78pt.”

6. Worktree discipline maintain karo.

When multiple worktrees exist, render and audit must print:

- worktree path
- branch
- commit
- dirty status
- binary path
- source path
- output path
- target

- preset

This prevents the exact kind of confusion where a fix exists in one worktree but render happens in another.

The deeper point is this: high quality PDF rendering is not just typesetting. It is a chain of reversible explanations. Jab bug dikhe, humko screenshot se source tak wapas jaana chahiye without guesswork.

That is the standard.

# Chapter 9: False Closure At A Split Table

A table is not only a grid of cells. A table is a promise: "Stay with me; these facts belong together." When that promise crosses a page boundary, the renderer has to carry the reader's attention across the gap.

The bug you noticed is subtle because the page is not technically lying. The table does continue on the next page. The row is split. The next fragment appears. But the top page paints a bottom rule, leaves white space, and visually relaxes. To the eye, that combination says: "This object is finished."

That is the problem of **false closure**.

At the perception level, closure is a Gestalt principle. The mind completes a shape and treats it as whole. A rectangle, a ruled box, a shaded area, a bottom border: these are not passive decorations. They are instructions to the eye. They say where a thing begins, where it ends, and whether the reader can move on.

When a split table paints its first fragment like a complete table, the page creates the wrong instruction. The document says, “continued,” but the visual grammar says, “closed.” The reader believes the visual grammar.

Formal names for this class:

- false closure
- premature closure
- missing continuation cue
- continued table ambiguity
- split-table continuation failure
- fragment boundary decoration mismatch
- continuation-aware decoration failure
- non-terminal fragment painted as terminal

In CSS and paged-media language, this lives near **box fragmentation** and **box decoration across breaks**. A block can be fragmented across pages. The hard question is whether each fragment should paint as if it were its own complete box, or whether non-final fragments should paint as open continuations of the same box.

For tables, “complete box” is often the wrong answer. If the table continues, the first fragment should not look finished. The bottom edge needs to communicate suspension, not completion.

There are several ways to do that:

- Extend the table background to the bottom of the usable content area.
- Suppress or soften the terminal bottom rule on non-final fragments.
- Use a continuation seam instead of a closing rule.
- Add an explicit “continued” marker when the table style allows it.
- Keep split-row backgrounds continuous across the break.
- Prefer row-level splitting only when the continuation is visually obvious.

The goal is not decoration. The goal is truthfulness. A table fragment should describe its state honestly. If it is the final fragment, it may close. If it is not final, it must not pretend to close.

This is why the white gap in the screenshot feels wrong. It is not merely unused space. It is a semantic pause in the wrong place. The eye sees the bottom rule, then the blank area, then the page number. The reader receives a complete ending. On the next page, the table returns, and the mind has to repair the story.

A good renderer should not make the reader do that repair.

The trace contract should therefore record the fragment phase:

```
table=node-143
fragment=0
phase=first
continues=true
terminal_decoration=false
continuation_background=
extends_to_region_bottom
```

```
split_row=true
split_row_continuation_marker=visible
```

The audit rule can be direct:

```
if table_fragment.continues:
    assert not paints_terminal_bottom_rule
    assert continuation_cue_present
    assert visual_background_does_not_imply_fin
al_closure
```

This is a small rule with a large effect. It changes a split table from a broken interruption into a continuous object.

The deeper principle is simple: page breaks are physical, but meaning is continuous. The renderer must respect both.

# Chapter 10: Avoidable Underfull Pages

The next issue is the large gap before the bottom of a page. The page has enough space to fit the beginning of the next text block, but the engine moves the entire block to the next page.

This is an **avoidable underfull page**.

It is not the same as every blank area. Books are allowed to breathe. Chapter openings may have deliberate space. Footnotes may reserve area. Images and tables may be atomic. A heading may need to stay with the first lines that follow it. But when a normal paragraph, heading bundle, or code listing

has a legal prefix that would fit, pushing the whole block is wasteful.

The formal names:

- avoidable page-fill gap
- underfull page
- loose page break
- missed partial-fragment opportunity
- unnecessary block deferral
- legal-prefix placement failure
- page badness not charged strongly enough

The word **legal** matters.

A renderer should not blindly fill every remaining point of space. That creates worse typography. One orphan line at the bottom of a page can be uglier than a gap. A heading alone at the bottom is worse than a gap. One lonely line of a code block may look like debris.

So the engine needs a legality test before it accuses a page break of being bad.

For prose:

```
remaining_space ⩾ height_of(min_first_fragment_lines)
```

If the preset requires at least two lines before a paragraph can split, then one line does not count as usable. Two lines count. Three lines are better.

For headings:

```
remaining_space ⩾ heading_height + keep_with_next_payload
```

A heading is a signpost. A signpost with no road after it is bad typography. The legal fragment is not the heading alone; it is the heading plus enough following content to prove that the section has begun.

For code:

```
remaining_space ⩾ height_of(min_code_fragment_lines)
```

Code has stronger chunking than prose. One line of code at the bottom often reads like an accident. A good default is usually two or three lines, depending on preset and code block density.

For tables and images, the rule changes. A table might be row-splittable, cell-splittable, or atomic. An image may be atomic. The page-fill detector should know the block type before judging.

The right audit finding should not say, “large whitespace.” That is too vague. It should say:

```
page=8
remaining_main_flow=58pt
next_block=paragraph node-812
legal_prefix=2 lines
legal_prefix_height=34pt
decision=deferred_entire_block
finding=avoidable_underfull_page
```

That finding is useful because it names the missed move. The engine had a legal action and did not take it.

There is a cost-model version too. Pagination is a negotiation between evils:

- widow/orphan violations
- stranded headings
- footnote collisions
- table fragmentation ugliness
- code block fragmentation ugliness
- page whitespace

Whitespace is not always evil. But avoidable whitespace should carry a cost. The cost should rise when the next block has a legal prefix that fits.

The solver should therefore ask a concrete question at every break:

“What is the smallest respectable piece of the next block, and does it fit?”

If the answer is yes, the page break needs a very good reason.

This gives us a clean invariant:

```
If remaining main-flow space can fit a legal
prefix of the next splittable block,
and no higher-priority keep, footnote, region,
or atomicity rule forbids it,
then the engine must not eject the whole block.
```

That is the whole idea. Not “fill every page.” Not “ignore beauty.” Just do not waste a page when a legitimate, readable continuation was available.

The human version is even simpler: do not make the reader turn the page before the page is done speaking.

# Appendix: Short Glossary

Visual symptom: Jo page par dikhta hai.

Running table header: Split table ke continuation page par repeated header.

Fragmentation: Ek block ko page-sized pieces mein todna.

Idempotence: Same input, same context, same output.

Derived state: Source se compute hua intermediate data.

Connascence: Hidden coupling jahan ek rule multiple places mein tied ho.

Invariant: Rule jo valid output mein hamesha true hona chahiye.

Metamorphic test: Test jo exact output ke bajay relation check karta hai.

Derivation trace: Decision history jo batati hai ki final mark kahan se aur kyun aaya.

False closure: A non-final fragment looks visually complete, so the reader thinks the object has ended.

Continuation cue: A visual signal that a fragmented object continues elsewhere.

Avoidable underfull page: A page leaves unnecessary blank space even though a legal prefix of the next splittable block could fit.

Legal prefix: The smallest typographically acceptable fragment of the next block, such as two paragraph lines, heading plus payload, or a minimum code fragment.