The Critical Path

How I Find the Limit Before the User Reaches It

Ioan Marian

MINI-BOOK · 30 PAGES

OPENING

I Wanted to See How Far It Would Go

An experiment at home, before the first client.

In 1998, I was working at home on an accounting application in Access. Before giving it to its first client, I did not check only whether it opened, calculated, and saved. I wanted to find out what happened as the data grew.

I gradually increased the volume. With a few thousand records, it worked well. With a few hundred thousand, it began to lose pace. Around one million, it stopped responding. I found the limit in my room, not the client after delivery.

I had begun measuring in 1994, on the university's 286 and 386 computers. The 1998 experiment left me with a question: how far will it go, and what happens when the volume grows? The mini-book can be read on its own. Each lesson explains the problem, the test, and the result.

> WHY
>
> **I do not optimize for an impressive number. I remove unnecessary work so the application remains fast, easy to understand, and easy to change as data and requirements grow.**

One test tells me that the application works. A volume test tells me how far it can go.

THE MINI-BOOK MAP

From effect to cause, then from solution to continuity

The 26 lessons are in the order in which I work: first I make the problem visible, then I eliminate the unnecessary work, and finally I leave a method that someone else can continue.

01

I Understand What Is Happening

Lessons 01-09
Incidents, logs, benchmarks, volume, and breaking points.

02

I Remove Unnecessary Work

Lessons 10 to 21
The critical path, ID-first, parallelism, search, and models.

03

I Build Continuity

Lessons 22 to 26
Preparation, progress, evaluation, protection, and transfer.

HOW TO READ IT You can go through all 30 pages, or you can start with the problem that is closest to your system.

LESSON 01 | UNDERSTAND WHAT IS HAPPENING

One line of code blocked an entire platform

It was lunchtime. I was about to take a break when the manager came to me, worried: 'The intranet application no longer works.'

I asked calmly: 'Which one?' I thought one isolated page had stopped temporarily. I was not expecting the answer: 'None of the pages work.'

The platform was used every day by many people. It brought together documents and several internal applications. It had worked for years without a similar incident.

A race against time began. In about fifteen minutes, the component causing the blockage was isolated. It had been developed several months earlier. We disabled it immediately, and the users could work again.

Only then did I open the source code. I knew the model the component was supposed to follow. When I opened the code, the line that broke that model caught my eye in about twenty seconds.

Data reading had not been kept as a separate stage. The code read a list, then each item inside the loop triggered more reads.

At first, the volume was small. The page worked normally, and nobody noticed the problem. Then the volume grew quietly. Each read seemed short, but repeating it for every item changed the total time completely.

I moved the read before the loop and reused the result. After testing in every environment, the corrected component returned to production the same day.

TRY THIS Choose an incident that is already visible. First restore the service; only then reconstruct the cause step by step.

IDEA

The platform was already blocked. I no longer needed to measure how slow it was. I needed to find what had stopped it.

LESSON 02 | UNDERSTAND WHAT IS HAPPENING

By tomorrow, make it four times faster

I was in the middle of an architecture discussion when my manager walked in. The request concerned a page I had never worked on. Another team had developed it, and I had inherited it without context or documentation.

At noon, the request was clear: 'By tomorrow, the first page must be four times faster. By this evening, I need the implementation steps.'

The page was not failing. It worked, but it was the home page of an internal SharePoint platform, it was used very often, and it took several seconds to load.

One XSL file - a template that turned data into the page seen by the user - handled every case and was called several times while the same page was being built.

I had several options. I could rewrite the display completely. I could keep the result temporarily in a cache, which would make the page appear faster without removing the cause. Or I could split the large file and run only the part that was needed.

I did not choose based on preference. In about ten minutes, I added timings to the log and measured the components separately. In another ten minutes, I moved the case selection into C# and split the original file into smaller XSL files. Each one was called once, only when needed.

I changed neither the rules nor the information displayed. I changed only how the work was divided. After testing in every environment, the solution was deployed the same day.

The result exceeded the target: the page became almost six times faster.

The pressure passed, but the lesson remained: sometimes the gain comes simply from dividing the work correctly.

TRY THIS	When time is short, measure the components and look for the smallest change that removes the most repeated work.
IDEA	**Pressure does not force me to change a lot. It forces me to choose precisely.**

LESSON 03 | UNDERSTAND WHAT IS HAPPENING

We no longer have the documents

I was asked to move a large document archive from an internal platform to another storage location. It contained procedures, internal documents, and archives built over years of work.

I downloaded the documents, organized them into folders, and placed them at the specified destination. Everything appeared to be fine.

Some time later, I learned that the documents were no longer available at either the source or the destination. The question was simple and difficult: did another copy still exist somewhere?

By chance, I had not yet deleted the temporary folders from the old server. That unintended copy made it possible to recover the entire archive.

I do not present this incident as a good procedure. I was lucky. If I had deleted the temporary folders immediately after the transfer, the documents could not have been recovered.

Some documents can be recreated in a few hours. Others can never be recreated.

Always make a backup.

TRY THIS

For an important transfer, verify the destination, the backup, and the restore separately before closing the source.

IDEA

A backup that has never been restored is still only a promise.

LESSON 04 | UNDERSTAND WHAT IS HAPPENING

Find a limit you can reproduce

Do not start with the whole system. Pick one operation and make the breaking point visible.

Choose a reading that you can repeat. Keep the same columns and increase the number of rows ten times at each step until the time exceeds the accepted threshold or begins to increase faster than the volume. Then go back to the same number of rows and change only how many columns you read. Do not change both axes in the same sample.

If you learn now	If you coordinate	If you design
A single change	Do not just ask for a number	Two axes

TRY THIS Choose one read and increase only the number of rows until the curve changes shape.

LESSON 05 | UNDERSTAND WHAT IS HAPPENING

The file name says what started and when

I use the pattern ClientList_YYYYMMDD_HHMMSS_FFF.txt. For example, ClientList_20260806_115522_012.txt shows that the ClientList operation started on August 6, 2026, at 11:55:22.012.

Each run receives its own file. This keeps two requests with different volumes, timings, or results from being mixed together. The name tells me the operation and the starting time. The content shows the steps in the order in which they occurred.

I record the start of each phase, its duration, the total time since the run began, the number of rows, files, or calls, and the business values needed to verify a calculation. At the end, I record the result and any error. I do not write personal data or sensitive content that does not help the analysis.

01	**ClientList**	The business operation that started.
02	**YYYYMMDD**	The year, month, and day when the run started.
03	**HHMMSS**	The hour, minute, and second when the run started.
04	**FFF**	The fraction of a second that distinguishes two runs started close together.

TRY THIS Create a new file for every important run and make its name clear enough to find it without opening the code.

IDEA

A good log does not merely say that the operation ended. It tells the full story of the execution.

LESSON 06 | UNDERSTAND WHAT IS HAPPENING

I observe several days or read the history that already exists

An isolated value becomes a question first, not an urgent optimization.

If the risk allows it, I observe for a few days and check whether the deviation repeats. If I already have two weeks of history, I can see immediately whether the trend was already present.

I keep a separate run history for every page and calculate p50, p95, and p99. I calculate the same values for the application as a whole, but the overall value can hide the degradation of one page. p95 is the time within which 95% of runs finished; the remaining 5% took longer.

If I have 1,000 comparable runs and p95 is 0.3 seconds, about 950 finished within 0.3 seconds and about 50 were slower. The analysis is both manual and automatic: daily tests and standardized reports show me the current state of every page and its trend over time.

Today One observation	**A few days** Repetition	**Two weeks** History
Periodically Intervention		

TRY THIS Calculate p95 for all runs and separately for each page. Then check whether the deviation repeats.

IDEA

I read the figure in context, confirm the trend, and act before it becomes an incident.

LESSON 07 | UNDERSTAND WHAT IS HAPPENING

The algorithm, input data, sorting, and result

I was not yet measuring SQL Server or milliseconds; I was learning how to make the problem verifiable.

In COBOL, I started with input data I knew, ran the calculation and sorting, and then checked the resulting report. In dBase IV and FoxPro, I kept the same order: known input, explainable steps, and a verified result.

My first benchmark was not a very small measured time. It was a case I could repeat and whose result I could verify after every change.

Input	Steps	Output
Controlled data	Calculation and sorting	Verified report

TRY THIS Build a simple table with the operation, volume, time, and verified result.

IDEA

My first benchmark was not a millisecond. It was a result I could verify.

LESSON 08 | UNDERSTAND WHAT IS HAPPENING

One million rows do not have a single cost

This is a separate SQL laboratory; it is not part of the automated C# series in the repository. I want to observe the effect of the number of rows and the number of columns separately.

1. I create a synthetic model with one million rows and fifty columns: ID, nineteen numbers, ten date values, ten titles, and ten descriptions.

2. I first select only the ID. I repeat the same test with 1, 10, 100, 1,000, 10,000, 100,000, and 1,000,000 rows.

3. I start again from one row for each group of 5, 10, 20, and 50 columns. I do not simply choose the first physical columns. Each group mixes numbers, dates, titles, and descriptions.

4. The group of 5 contains ID, two numbers, one date, and one title. The group of 10 adds more numbers and dates, another title, and two descriptions. The group of 20 has four descriptions. The group of 50 includes all ten descriptions.

5. The descriptions have different lengths, so ten columns can move much more data than five, and the increase does not just follow the number of columns.

6. The table shows the results for the volume of one million rows. Volumes are reproducible estimates of synthetic load. SQL times depend on the server, indexes, cache, and network and must be measured in the environment in which the database runs.

SELECTION	RETURNED VALUES	ESTIMATED VOLUME (MB)
Only ID	1,000,000	8.00
5 columns	5,000,000	77.00
10 columns	10,000,000	478.00
20 columns	20,000,000	2,820.00
50 columns	50,000,000	7,170.00

TRY THIS Create a test table with short and long text. Compare the same rows while selecting 1, 5, 10, 20, and 50 columns.

IDEA

One million rows do not have a single cost. The amount of data carried by each row matters too.

LESSON 09 | UNDERSTAND WHAT IS HAPPENING

The same useful work, from one instance to a billion new instances

This is an indicative calculation, not a benchmark included in the public manifest. I preserve exactly one billion updates and show how allocation changes when the new instruction is moved inside the repeated operation.

1. I assume a small class for which an object occupies 24 bytes, the same size seen in the public demo of ten million objects.

2. I run one billion useful updates in every version. With one object, it receives all the updates. With 1,000 objects, each receives one million. With one billion objects, each receives one update.

3. The memory column is calculated: the number of objects multiplied by 24 bytes. One billion objects produce approximately 24 GB of cumulative allocations.

4. I do not publish execution times here. They must be measured on the reader's computer, because the JIT, the processor, and the garbage collector can significantly change the result.

CREATED INSTANCES	UPDATES / INSTANCE	Calculated memory (MB)
1	1,000,000,000	0.000024
1,000	1,000,000	0.024000
10,000	100,000	0.240000
1,000,000	1,000	24.000000
1,000,000,000	1	24,000.000000

TRY THIS

Keep one billion updates fixed and change only the number of objects created. Compare the time and allocated memory.

IDEA

The number of loop iterations does not tell me how many instances are created. The position of the new instruction decides that.

LESSON 10 | REMOVE UNNECESSARY WORK

The first version read the entire set for a single page

The historical example uses approximate values: 60 columns, 100,000 rows, and a few dozen displayed rows.

The first version returned about 60 columns for about 100,000 rows. That meant nearly 6,000,000 values for a single page. I changed the order of operations. I selected the IDs first, kept only the current page, and only then read all the fields for those rows.

With two reads, I returned about 100,000 IDs and 3,000 values for the current page: 103,000 values in total, compared with 6,000,000. With three reads, on page 6, I had a COUNT, about 300 IDs, and 3,000 values. The total was 3,301. A very deep page required more IDs, so the advantage had to be measured again.

These ratios describe the information returned to the application; they do not claim that SQL Server's internal work automatically decreases by the same proportion. The filter, index, and sort determine how much the database must read and order. ID was the last sort criterion, so equal values in the other fields could not change row order from one request to another.

6,000,000 Original Form	**103,000** Two reads	**3,301** Three reads, page 6

TRY THIS Draw the path of a page in four boxes: reading, processing, dependencies, and display.

LESSON 11 | REMOVE UNNECESSARY WORK

The same page, but 7.65 GB, 4.38 MB, or 0.38 MB

This is another separate SQL laboratory, with wider synthetic rows than in the previous test. I want to display the same fifty rows from a large table. I compare three paths and verify that the final page is identical.

1. I create one million rows. Each row has fifty columns: an ID, numbers, dates, titles, and descriptions up to 800 characters long.

2. In the first path, one SELECT - the SQL instruction that reads data - returns all fifty columns for every row. The application receives the entire set and keeps only the fifty rows of the current page.

3. In the second path, I select all ordered IDs. I take the IDs for the current page and run another SELECT for all columns only for those fifty IDs.

4. In the third path, I run COUNT for the total number of rows, select only the IDs needed up to the requested page, and read all columns only for the fifty displayed rows.

5. I calculate the same control value from every field on the page. All three paths return the same page; only the work required to reach it differs.

6. This laboratory reproduces the historical path in three reads: COUNT, the IDs through the end of the page, and the full rows of the page. It does not use the narrow index and Skip/Take from the next laboratory. That is why page 6 has different times: the conditions differ, not the result.

VARIANT	RETURNED VALUES	VOLUME (MB)	TIME (seconds)
The whole set: 50 columns	50,000,000	7,650.00	21.00
ALL IDs + page	1,002,500	4.38	0.63
COUNT + 300 ID + page 6	2,801	0.38	0.10
COUNT + 62,850 ID + page 1,257	65,351	0.63	0.15

TRY THIS

Repeat the experiment with wide rows. Compare the full set, all IDs, and only the IDs needed up to the requested page.

IDEA

I am not interested only in how many times faster one approach is. I want to know whether the final time remains low as the volume grows.

LESSON 12 | REMOVE UNNECESSARY WORK

The number of reads was a measured decision

I chose the version that read and transferred less data.

In the SQL books and examples I studied then, pagination was usually shown with a single query. I did not begin with a known two- or three-read model. I reached it by testing the ordered IDs, the rows of the current page, and COUNT - the operation that counts all matching rows - separately.

With three reads, I first execute COUNT to obtain the total. Then I use TOP to select the ordered IDs needed through the end of the requested page. From this list, I keep the IDs of the current page and, in the third read, load every column only for those rows.

If a difficult WHERE filter already reduced the result to a few thousand rows, two reads could be better than three. If the user needed the total count, I measured it separately. I did not turn the architecture into a rule that could never change. I tested one, two, and three reads on the same data and chose the lowest total cost.

01	**I read once**	Filter, order, page, and fields in an efficient plan.
02	**I read twice**	IDs, then objects; the intermediate set is small.
03	**I read three times**	COUNT, IDs, and objects are justified costs.
04	**I measure**	I choose from the measured data, not from the number of commands.

TRY THIS Measure the versions with one, two, and three reads separately. Keep the version that moves the least data.

IDEA **The number of queries was secondary. The goal was to read less data.**

LESSON 13 | REMOVE UNNECESSARY WORK

Nine seconds became 0.17 seconds, but the absolute time tells the better story

This is a separate SQL laboratory, not a numerical continuation of the previous table. I rebuild ten million wide rows with a different synthetic distribution and a different SQL plan, then request the same page through two paths.

1. I create a synthetic table with ten million rows, fifty columns, and ten descriptions of 50 to 200 characters.

2. A page has fifty rows. I sort by the selected field and then by the ID, so that the same request always produces the same order.

3. I create a narrow index that only contains the sorting field and the ID.

4. In the usual variant, I select all fifty columns and apply Skip/Take directly to them.

5. In the ID-first variant, I apply Skip/Take only to the ID. Then I select all the columns for only the fifty IDs on the page.

6. I verify the same fifty rows, the same 2,500 values, and the same control value.

7. I measure the complete time and the logical reads. A logical read is a data page examined by SQL Server, not a row returned to the application.

8. I repeat each version in three independent processes and keep the representative time.

PAGE	ROWS SKIPPED	ALL COLUMNS (seconds)	COMPLETE ID-FIRST (sec)	RATIO
1,257	62,800	0.1660	0.0040	≈ 40×
10,000	499,950	1.3800	0.0240	≈ 57×
50,000	2,499,950	6.5500	0.1200	≈ 55×
70,000	3,499,950	9.2000	0.1670	≈ 55×

TRY THIS

Compare Skip/Take on every column with Skip/Take only on the ID, followed by reading the rows of the current page.

IDEA

Even after a large gain, I check whether the final time has fallen far enough for the next increase.

LESSON 14 | REMOVE UNNECESSARY WORK

The same 101,000 rows: one second or one third of a second

This is a third separate SQL laboratory, with twenty columns and a different synthetic workload. I want to see whether I can reduce read time without changing the data. Modulo 4 means that I divide the rows according to the remainder when the ID is divided by 4; the remainder can only be 0, 1, 2, or 3, so I obtain four non-overlapping groups.

1. I create a small table with 1,000 rows and a large table with 100,000 rows. Each row has twenty columns: numbers, dates, titles, and descriptions. In total, I read 2,020,000 values, about 307.55 MB.

2. In the first version, I read the small table and then the large table. These are two complete queries executed one after the other.

3. For the other two versions, the small table remains the first task. I split the large table into four groups by the remainder of the ID divided by 4: 0, 1, 2, and 3. Each row belongs to exactly one group.

4. I first run the five tasks one after another. This test shows the cost of splitting the work without the benefit of parallel execution.

5. I then run the same five tasks at the same time, on separate connections.

6. I combine the rows, order them by source and ID, and calculate the same value from every field. All versions return the same 101,000 rows.

VARIANT	TASKS	ROWS	VOLUME (MB)	TIME (seconds)
Two complete reads, serial	2	101,000	307.55	1.02
The same five parts, serial	5	101,000	307.55	1.10
The same five parts, parallel	5	101,000	307.55	0.35

TRY THIS

Split the large table into four groups with modulo 4. Run them first serially, then in parallel, and verify the same result.

IDEA

Modulo 4 gives me four clear groups. Measurement tells me how much I gained. In this test, the same read finished in about one third of the time.

LESSON 15 | REMOVE UNNECESSARY WORK

The same 300,000 results: 16.035200 seconds or 0.015229 seconds

I change only the path to the appropriate child object. The calculation applied to each result and the final value remain the same.

1. I create four levels. The first has 300 objects. Each object has ten children, so the next levels contain 3,000, 30,000, and 300,000 objects.

2. The 300,000 objects on the final level are the useful results. I perform the same calculation for each one.

3. In the LINQ/Where variant, for every parent I scan the entire next-level list again and check which objects belong to it.

4. In the array variant, I go directly to the positions of the matching children. I no longer rescan the entire list for every parent.

5. LINQ/Where performs about 9.09 billion checks. The array visits only the 333,000 required objects.

6. Both variants produce 300,000 results and the same final sum. I only change the path to the child object.

VARIANT	CHECKED ELEMENTS	AVERAGE TIME (seconds)	RATIO	RESULTS
LINQ/Where repeated	9,090,900,000	16.035200	1×	300,000
Array with direct access	333,000	0.015229	≈ 1,053×	300,000

TRY THIS

Build the same relationships with LINQ/Where and with an array of positions. Count the checks, not only the seconds.

IDEA

The search can cost more than the operation performed after the result has been found.

LESSON 16 | REMOVE UNNECESSARY WORK

25,000 terms in 100,000 documents: about 41 hours or 26 minutes

I want to find 25,000 terms in a large document. For one term, IndexOf is the simple and natural solution: it scans the text and reports where the word appears. Repeated for 25,000 terms, the same solution scans the document 25,000 times.

1. I create a synthetic document of about 200 pages, containing 1,105,000 characters.

2. I create 25,000 terms. Both methods must find exactly the same 100,000 matches.

3. IndexOf is the straightforward solution: I search the entire text for the first term, then the second, and continue through the 25,000th term. The document is scanned again for every term.

4. Aho-Corasick is the optimized method for many simultaneous searches. I build one map of the terms, then scan the document once. The map takes approximately 0.006374 seconds to build and can be reused.

5. The table projects the measured time for one document across 100,000 similar documents. It does not claim that I ran the test for 41 hours.

METHOD	ONE DOCUMENT (seconds)	100,000 DOCUMENTS (seconds)	TIME IN FAMILIAR UNITS	RATIO
Repeated IndexOf	1.479386	147,938.60	Approximately 41 h 6 min	1×
Aho-Corasick	0.015784	1,578.43	approximately 26 min 18 s	≈ 94×

TRY THIS Build the Aho-Corasick map once, then compare the same matches with repeated IndexOf searches.

IDEA

When I look for 25,000 things, I try not to go through the same document 25,000 times.

LESSON 17 | REMOVE UNNECESSARY WORK

When several threads on the same computer do not solve the problem

A thread is one path of work executed by a computer. The starting point is a video-processing job in which one thread runs the tasks one after another on one computer. The estimate is twelve days.

1. In the initial version, one computer processes the tasks one after another on a single thread. The estimated time for the entire batch is twelve days.

2. I try several threads on the same computer. The gain is not enough because the tasks share the same processor, memory, and local limits.

3. The video tasks are independent. A colleague suggests Azure Batch, a service that can start several computers and send one task to each of them.

4. I split the batch, keep an identifier for every task, and combine the results at the end. A failed task can be retried without repeating the entire batch.

5. The table shows the ideal mathematical division of the twelve days. In real execution I add computer startup, transfer, waiting, retries, service limits, and financial cost.

SIMULTANEOUS TASKS	COMPUTERS	IDEAL LIMIT	WHAT IT MEANS
1	1	12 days	starting point
50	up to 50	5 hours and 46 minutes	ideal sharing
100	up to 100	2 hours and 53 minutes	ideal sharing
200	up to 200	1 hour 26 min	ideal sharing

TRY THIS

Before distributing the work, verify that the tasks are independent and that their results can be safely combined and retried.

IDEA

Sometimes I do not have to force the same computer. I have to divide the problem into units that can run on different computers.

LESSON 18 | REMOVE UNNECESSARY WORK

One million lines and two solutions that cost too much

A PowerShell script had to create a file of about one million lines. The file was necessary; the problem was how I was building it.

The first version wrote every line right away. The result was correct, but it produced about a million file operations. Opening, writing, and repeatedly syncing became the dominant cost.

The second variant kept all lines in a single string. In PowerShell, strings are immutable: each append creates a new string, copies the entire existing content, and then adds the next line. For four units, the work is 1 + 2 + 3 + 4 = 10, even though the final result contains only four units.

I needed the same final file, but without a million writes and without repeatedly rebuilding a single giant text.

Line by line	One text	Target
≈ 1,000,000 writes	History is copied	Same final file

TRY THIS Build the same text with all four methods. Verify the same hash, then compare time and cumulative allocation.

IDEA

I wanted the same file without one million writes and without repeatedly rebuilding one huge string.

LESSON 19 | REMOVE UNNECESSARY WORK

Ten records said yes. One million said no

In February 2004, Web Forms and Web Controls were the usual path in ASP.NET. Visual components and events helped developers build the first page quickly.

I built a test in C#. With ten records, the page worked well. With one million, it froze. The large volume showed me that I did not want to continue in the same direction.

I did not try ID-first pagination inside the Web Controls page. In about five minutes, I decided to build a different engine around the business object, context, and action.

The sketch was cli;list;detail;save;5. cli meant the Client object. list was the screen I wanted to reach. detail was the screen from which the request came. save was the action. 5 was the ID of the existing client. For creation, I used ID -1.

The line was not the finished engine. It was its map. The same stages could be used for Client, Invoice, or User. In my own model, I applied ID-first pagination, and the one-million-row test fell below one second.

cli Client object	**list / detail** Destination and origin	**Save / 5** Action and ID

TRY THIS Take an existing page and write its object, context, action, and identifier on one sheet.

IDEA

I did not build the engine in five minutes. I chose the direction in which I would build it over the next nine months.

LESSON 20 | REMOVE UNNECESSARY WORK

A change had to be made once, not on every page

Web controls were ASP.NET components that helped build a web interface quickly. The first page was easy to write; the question was what happens when the model reaches dozens of pages.

With Web Controls, I could quickly build the first page, but each new page started with a lot of work of its own. If a common rule changed, I had to look for all the places where it had been repeated. My engine kept the common steps in one place and left the specific configuration and behavior to the app.

The application configurations lived in XML files - structured text files that described the differences. I did not load web.config, the application's general configuration file, with the details of every object. This let me see clearly what belonged to the engine, what belonged to the application, and what had to change when a new type of page appeared.

01	**3 pages**	The duplication is hidden; the cost seems acceptable.
02	**40 pages**	Duplication spreads across multiple places.
03	**The engine**	Common steps stay in one place.
04	**XML**	The application declares the differences without rewriting the engine.

TRY THIS Choose three similar classes and note what can be shared without hiding their real differences.

IDEA

The problem was the number of places where the same decision had to be repeated, not the number of pages.

LESSON 21 | REMOVE UNNECESSARY WORK

AI can build and multiply a good model very quickly

Once the first example is clear, similar classes can be created much faster.

I can give the AI a complete class, the conventions, the tests, and the differences of the new object. It can quickly propose Client, Invoice, or User on the same structure. It can complete validations, logs, and repetitive tests and compare implementations.

The model gives me speed because I do not make the same decisions again from scratch. It gives me confidence because logging and tests are already part of the example. It makes the result easier to predict because the next object follows familiar steps. AI increases these advantages, but it can multiply a mistake just as quickly.

TRY THIS	Before generating the second class, write down the rules that the first class has already demonstrated.

IDEA	**AI multiplies the model. I still decide what is worth multiplying.**

LESSON 22 | BUILD CONTINUITY

Thirty minutes for a few people. Much more for the entire construction site

The thirty minutes did not go away. They came back in the form of harder and more expensive work.

At the moment of decision, the situation seems small and personal: do we finish the break, or do we get up now? But the consequence is not measured only in each person's time. It appears in the work of the entire team, the lost concrete, transportation, the construction delay, and the work that must begin again.

This idea has remained in my way of working. I look at the small decision first. Then I put together all the effects it has on the team and the system. What seems cheap in one place can become very expensive after I count all the new steps.

TRY THIS

Choose a short action that blocks other activities and note the cost of postponing it for one day.

IDEA

The thirty minutes saved moved the work into a much harder and more expensive form.

LESSON 23 | BUILD CONTINUITY

From 0.25 seconds to almost 40 seconds

A small cost, multiplied weekly, does not stay small.

I start at 0.25 seconds and add five percent each week. After 104 weeks, the mathematical result is about 40 seconds. In a real application, the curve will not be this regular. Some weeks bring no change; others bring a jump. The example shows how easy it is to ignore a loss that looks tolerable in isolation.

Time can increase for many reasons. There may be more data, a new loop, an external call, or a missing database index. If I see only the final time, all these causes are mixed into the same sentence: the page is slow.

TRY THIS Choose one page and record its p50, p95, volume, and application version every week.

IDEA **Five percent does not look alarming on a Friday. Repeated every week for two years, it changes the nature of the problem.**

LESSON 24 | BUILD CONTINUITY

The fact that it works today is only the starting point

A demonstration confirms the function today. The history shows how the application changes.

I start with what is there: users, rows, pages, rules, integrations, timing, and errors. Then I look at the history. How fast did the data grow? How many exceptions have emerged? How many places must change for a new rule? How much time belongs to the system and how much to external dependencies?

I do not mechanically extrapolate from a single month. I compare periods, volumes, and types of execution. I normalize time by the processed volume before saying that the system has become slower. If the number of rows remains stable and time rises, I look for a slower version, a new path, or a dependency that has changed.

Today Current state	**History** Change in time	**Tomorrow** Scenario

TRY THIS Take a function that works today and write three growth scenarios that you can test.

IDEA

I compare time with volume and build scenarios from the history.

LESSON 25 | BUILD CONTINUITY

The Protection Could Be Forgotten

The test code could modify real data, and stopping it depended on a manual step.

I used this practice because it provided access to real cases missing from the test data. I ran a small number of carefully chosen examples and saved time. Successful executions created the impression that the method was under control.

One day, while tired, I forgot the protective step. The code performed operations where it should not have, and real data had to be restored. The problem was not that I consciously chose to delete data. The problem was that the system did not refuse the operation when the context slipped from my attention.

01	**I gain immediately**	I quickly test real and varied cases.
02	**I repeat without incident**	Practice starts to seem normal.
03	**I forget the safeguard**	Attention can fail on an ordinary day.
04	**I pay the price**	The operation reaches the actual data.

TRY THIS Turn a rule that depends on attention into an automatic check that can refuse the run.

IDEA

If safety depends on my remembering every time, I have no protection. I have only a hope.

LESSON 26 | BUILD CONTINUITY

A system that someone else can continue keeps the method, not the person

I do not want the next problem to start with the question, “Where is the author?”

The log preserves the execution. Tests preserve the result. Benchmarks preserve what we know about the limit. The model preserves the separation between common and specific. Documents preserve decisions. Examples show how to add the next object. None of these replaces people, but together they allow people to work without depending on one person’s memory.

This is the natural end of the critical path: not only a faster page, but a way of working that can be seen, verified, and passed on. If the system can continue without me, my contribution does not disappear. It becomes part of the way the system explains itself and evolves.

TRY THIS

Ask a colleague to explain the path through a function without your help, and note what is missing from the model.

IDEA

Another person can operate, investigate, and modify the system without making the author an indispensable dependency.

BEFORE THE NEXT CHANGE

Ten questions that keep the road clear

I do not need all the answers before I start. I need a question clear enough for the next test.

01 **What functional result must remain identical?**	02 **How long does it take now, and what is the volume?**
03 **How many times does the operation repeat?**	04 **Where does the work multiply?**
05 **What is the first breaking point?**	06 **What does my log show in production?**
07 **What work can I eliminate, not just accelerate?**	08 **How do I verify that the result remains correct?**
09 **What happens if the volume increases tenfold?**	10 **Can someone else continue without relying on my memory?**

THE CRITICAL PATH

I start with simple things, measure them, observe what happens as the data and repetitions grow, and change only what I can verify. Then I leave the method clear enough for someone else to continue it.

PUBLIC CODE AND RESULTS

https://github.com/IoanMarian2001/DemoCriticalPath
Exact version used by the book: book-v1.0.0