How the WFM Tools Page Actually Works

ERLANG FORMULAS
A PLAIN-LANGUAGE EXPLAINER

How the calculators on the Tools page do what they do — written for a curious fifteen-year-old.

ABL Digital Technologies

ERLANG FORMULAS
A PLAIN-LANGUAGE EXPLAINER

How the calculators on the Tools page do what they do — written for a curious fifteen-year-old.

ABL Digital Technologies

1 · The Problem in One Sentence

You run a phone line. Calls come in. You need to know how many people you have to sit by phones so that almost every caller gets through quickly, without paying for more staff than you need. That is the entire job.

On the surface this is simple — count the calls, multiply by how long each one takes, divide by an hour. But the calls do not arrive in a neat line. They arrive in clumps. Sometimes three at once. Sometimes none for two minutes. The clumps are what make the maths interesting, and the clumps are what the Erlang formulas were invented to handle.

Think of it like a school canteen at lunchtime. The headmistress knows that on average 200 children come in over the 40-minute lunch break. She could say: that is 5 children per minute, I will hire enough serving staff to handle 5 children per minute, problem solved. But every dinner lady knows this is nonsense. The children do not arrive one every twelve seconds in a polite line. They arrive in waves — when a class finishes, twenty-five children show up at the counter at once. A staffing plan built on the average will fail every single lunchtime. A staffing plan built for the waves will sit half-idle in the gaps. The right answer is somewhere in the middle, and finding it requires a little bit of clever maths. That clever maths is what this document is about.

1A · Some Words Before We Start

This whole document is meant to be readable by someone who has never seen any of this before. So before we go further, here are the few pieces of vocabulary we will need. If any of these are already obvious to you, skim past them. If not, this section is your friend — every one of these words shows up later.

Probability

A probability is just a number between 0 and 1 that tells you how likely something is. Zero means it never happens. One means it always happens. 0.5 means it happens half the time. We sometimes write it as a percentage instead: 0.5 is the same as 50%. So when we say "the probability of being blocked is 0.07," that is the same as saying "7 out of every 100 callers will hit a busy tone."

Average

Add a bunch of numbers up, divide by how many there were. If three callers wait 10, 20 and 30 seconds, the average wait is (10+20+30)/3 = 20 seconds. Everywhere we say "average" in this document, that is what we mean. Averages hide a lot — half of callers are doing better than the average and half are doing worse — but they are still useful.

Per hour

Almost every input on the Tools page is "per hour." That means: count how many of the thing happen during one full hour, then use that number. If you have 50 calls in 30 minutes, that is 100 calls per hour. If you have 200 calls in two hours, that is 100 calls per hour. Simple — but it is the unit that everything else is built on, so it matters.

Seconds in an hour

There are 60 seconds in a minute and 60 minutes in an hour, so an hour contains 60 × 60 = 3600 seconds. You will see the number 3600 pop up in the very first formula. That is all it means: the number of seconds in an hour.

AHT — Average Handle Time

This is the average time, in seconds, that an agent spends on one call. It includes the actual talking, plus any paperwork the agent has to do afterwards (in WFM jargon, "after-call work"). If your agents talk for an average of 3 minutes 30 seconds and then need 30 seconds of typing afterwards, AHT is 4 × 60 = 240 seconds.

SLA — Service Level Agreement

A target written as: "X% of calls answered within Y seconds." The classic target is 80/20 — eighty percent of calls answered within twenty seconds. So if a contact centre handles 1000 calls and meets the 80/20 SLA, that means at least 800 of those 1000 calls were picked up in 20 seconds or less. The other 200 may have waited longer, but the target only cares about the 80% threshold.

Factorial — the exclamation mark

You will see things like 5! or N!. The exclamation mark is the maths symbol for factorial. It is not surprise. It means: multiply every whole number from 1 up to that number. So 5! = 1 × 2 × 3 × 4 × 5 = 120. And 4! = 1 × 2 × 3 × 4 = 24. And 0! is defined to be 1 (it looks weird, but it is a maths convention that makes other formulas work out cleanly). Factorials grow ridiculously fast: 10! is already 3,628,800. 20! is about 2.4 quintillion. This is why we have to be careful with them later.

The number e

The letter e is a special number, just like π (pi) is a special number. It is approximately 2.71828. It comes up naturally whenever things grow or shrink smoothly over time — radioactive decay, cooling tea, compound interest, bacteria growing. You do not need to know where it comes from. You only

need to know that "e to the power of minus something" — written e^(−x) — is a number between 0 and 1 that starts at 1 when x = 0 and shrinks toward 0 as x gets bigger.

Exponential decay

Picture a cup of tea cooling. It starts hot — say 90 degrees. It cools fast at first because the gap between the tea and the room is big. As the tea gets closer to room temperature, it cools more slowly. After a long time it is essentially at room temperature and barely cooling at all. The graph of temperature over time has a characteristic shape — fast drop, then slowing curve, then flat. That shape is called exponential decay. e^(−x) draws exactly that shape. In our formulas, the thing decaying is "the chance you are still waiting in the queue after T seconds."

Recursion

A recursion is a recipe that uses the answer to a smaller version of itself to get the next answer. The classic example: to climb a staircase, climb one step, then climb the rest of the staircase. The "rest of the staircase" is a smaller version of the original problem. We will use a recursion to compute the Erlang B formula without ever calculating any huge factorials directly. We start at B(0), then use B(0) to find B(1), then use B(1) to find B(2), and so on, climbing one rung at a time until we get to where we want to be.

Erlang

A unit of telephone traffic load. One Erlang means one phone line is busy for one hour. Two Erlangs means either two phone lines busy for an hour, or one phone line busy for two hours, or anything in between. We unpack this properly in Section 2 — but every time you see the letter A in any formula on the Tools page, "A" stands for "the load measured in Erlangs."

> *Tip for reading the rest of this document: if you are stuck on a formula, do not try to understand it as a symbol. Try to picture what it is asking. Every Erlang formula has a simple, almost common-sense, idea hiding inside it. The symbols are just the shorthand mathematicians use to write that idea down quickly.*

2 · What an "Erlang" Actually Is

An Erlang is a unit of load. One Erlang means: one phone line is busy for an entire hour. That is it.

Let us slow that down. Imagine you watch a single phone for one hour. If that phone was busy for the whole hour, the load on it was exactly 1 Erlang. If it was busy for only half the hour, the load was 0.5 Erlangs. If you watched two phones and both were busy for the whole hour, the load was 2 Erlangs. The Erlang is just "hours of busy phone, measured per hour of clock time." Nothing more.

If you have a hundred calls per hour and each one lasts three minutes, add up the talking time:

```
Total talk time per hour = 100 calls × 3 minutes = 300 minutes
In hours: 300 / 60 = 5 hours of talking
```

So your load is 5 Erlangs. Five phones, each busy for the full hour, would exactly handle that amount of conversation. The formula the calculator actually uses is the same idea in seconds:

```
Traffic A = (Calls per hour × AHT in seconds) / 3600
```

Whenever you see the letter A in any Erlang formula, this is what it means: how many hours of work are arriving every hour.

Let us check that the formula matches the example we just did. We said 100 calls per hour, each lasting 3 minutes. 3 minutes is 180 seconds. So:

```
A = (100 × 180) / 3600 = 18000 / 3600 = 5 Erlangs
```

Same answer. Good. The reason we divide by 3600 is just to convert seconds back into hours — there are 3600 seconds in an hour, so dividing by 3600 turns "seconds of talking" into "hours of talking." That is the only piece of arithmetic happening there.

One more way to picture it. If you have 5 Erlangs of load, that is the same as saying: at any random instant during the hour, on average, 5 conversations are in progress somewhere. Not exactly 5 — sometimes 3, sometimes 7 — but 5 on average. So you obviously cannot survive with fewer than 5 agents on the phones. You need at least 5 just to keep up with the average. The whole rest of this document is about figuring out how many extra you need on top of those 5 to handle the moments when 7 calls are in progress at once.

> *Why the name? Agner Krarup Erlang was a Danish mathematician working at the Copenhagen Telephone Company. He published the original papers between 1909 and 1917. The unit was named after him by international agreement in 1946.*

3 · Why Five Phones Are Not Enough for Five Erlangs

Here is the trap. The load is 5 Erlangs. Surely 5 phones will do? No. Five phones will do, on average, but for half the hour you will be over-running and callers will get a busy tone or sit in a queue.

Calls do not arrive politely. Picture a school playground. Children do not line up at the water fountain every twelve seconds. They arrive in gangs, then nobody comes for two minutes, then another gang. Phone calls do exactly the same.

Mathematicians describe these arrivals with a thing called a Poisson process. The detail of Poisson is not important here. What matters is this: random arrivals always clump, and clumping creates moments when demand briefly spikes above the average. Your phones have to be sized for those spikes — not for the average.

The Erlang formulas are the tool that converts "average load" into "how many phones do I actually need." That conversion is the whole point.

A really concrete way to see this. Suppose you flip a coin once every second for a thousand seconds. On average you will get 500 heads. But in any given stretch of 10 seconds, you might get 3 heads, or 7 heads, or even all 10 heads. The clumping happens because each flip is independent of the last. Phone

calls work like that too: the fact that no call has come in for two minutes does not in any way "use up" the next call. It can arrive any second, including right now, including alongside two others.

This independence — that one call has nothing to do with the timing of the next — is what mathematicians call "memoryless" arrivals. It is the central assumption of the whole Erlang model. The arrivals have no memory of when the last call came in, and so they cluster randomly. This is what makes capacity planning hard, and it is what makes Erlang necessary.

Here is the punchline of this section, in one sentence: if calls arrived perfectly evenly, like a metronome ticking, then average load A would also be the exact number of agents you need. They do not arrive that way. So you need more than A. The Erlang formulas tell you exactly how many more.

4 · Erlang B — When Blocked Callers Just Go Away

4.1 The Setting

Imagine an old-school phone exchange. There are N trunks (lines). If a call comes in and all N lines are busy, the caller hears a busy tone and hangs up. Gone. No queue.

Erlang B answers the question: given the load A and N lines, what is the probability that a new call will hit the busy tone?

Let us name the pieces clearly so we never get lost:

- N = the number of phone lines (trunks) you have. A whole number, like 8 or 50 or 200.
- A = the load measured in Erlangs (we covered this in Section 2). Often a decimal — 5 Erlangs, 12.4 Erlangs.
- B(N, A) = the probability of being blocked, written as a number between 0 and 1. 0.07 means 7 callers out of every 100 hit a busy tone.

When you see B(N, A), read it out loud as "B of N and A" — a function that takes two inputs (how many lines, how much load) and gives back one output (the probability of blocking).

4.2 The Famous Formula

The original closed-form Erlang B formula looks scary:

```
B(N, A) = A^N / N!
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
A^0/0! + A^1/1! + ... + A^N/N!
```

Translation: take A to the power N, divide by N factorial (the product 1×2×3×…×N), and divide that by a sum of similar terms for every smaller number of busy lines. The numerator is the probability that exactly all N lines are busy; the denominator normalises it across every possible "how many lines are busy right now" state.

Let us slow that down. The two symbols you might not have met before are "A to the power N" (written A^N) and "N factorial" (written N!).

"A to the power N" is short for: multiply A by itself N times. So A^3 means A × A × A. If A = 5, then A^3 = 5 × 5 × 5 = 125. And A^0 is defined to be 1 (this is a maths convention; just accept it).

"N factorial" we covered in the vocabulary section: 1 × 2 × 3 × … × N. So 3! = 1 × 2 × 3 = 6.

Now the formula. The top piece, A^N / N!, is the relative likelihood that exactly N callers are on the line right now. The bottom piece is a sum that lists every possible state: zero callers, one caller, two callers, all the way up to N callers. By dividing the top by the whole sum, we are asking: "out of all the moments in time, what fraction of them is the system in the all-busy state?" That fraction is the blocking probability.

Why does that translate to "what fraction of callers are blocked?" Because if the system is in the all-busy state 7% of the time, and calls arrive randomly through the hour, then about 7% of arrivals will arrive during one of those all-busy moments. That is the link between "state probability" and "blocking probability." It is one of the small leaps of faith you have to make to follow the maths — but it is true, and Erlang proved it carefully in his original 1917 paper.

4.3 Why the Calculator Does Not Use That Form

The factorials and the powers get huge very quickly. With 200 trunks, 200! is a number with 375 digits. Computers handle that badly. So in practice everyone uses a recursion. The recursion is what the calculator on the Tools page actually runs:

```
B(0, A) = 1

B(k, A) = ( A × B(k-1, A) ) / ( k + A × B(k-1, A) )
```

You start with B(0, A) = 1 (if you have zero trunks, every call is blocked) and then you climb the ladder one trunk at a time, plugging the previous answer into the next step. After N steps you have B(N, A) — the blocking probability you wanted. No factorials. No huge numbers. A pocket calculator could do it.

Let us read that recursion as plain English.

The first line says: if you have zero phone lines, the probability that a call is blocked is 1 (i.e. 100%). That is just common sense — with zero lines, every call gets a busy tone. So we start there.

The second line says: to find the blocking probability with k lines, you take the answer for k−1 lines, do a small piece of arithmetic on it, and out pops the answer for k lines. The arithmetic is:

- Top of the fraction: multiply the previous answer by A.
- Bottom of the fraction: take the number k (how many lines you are at right now), and add A times the previous answer.
- Divide top by bottom. That is your new answer.

That is one rung of the ladder. To go from 0 lines to 8 lines, you climb eight rungs — eight applications of that arithmetic. At the top of the ladder you have your blocking probability for 8 lines.

Where did this recursion come from? It is the closed-form formula, rearranged. Someone (almost certainly Erlang himself) noticed that B(k) and B(k−1) are related by exactly the simple expression above, so you never need to compute the factorials or the powers directly. You only need the previous answer. That is the magic.

4.4 A Worked Example

Suppose A = 5 Erlangs and N = 8 trunks.

```
B(0) = 1
B(1) = (5 × 1) / (1 + 5 × 1) = 5 / 6 ≈ 0.833
B(2) = (5 × 0.833) / (2 + 5 × 0.833) ≈ 0.676
B(3) ≈ 0.530
B(4) ≈ 0.398
B(5) ≈ 0.285
B(6) ≈ 0.192
B(7) ≈ 0.121
B(8) ≈ 0.070
```

So with 5 Erlangs of traffic and 8 trunks, about 7% of calls hit a busy tone. Type "8" and "5" into the Erlang B calculator on the Tools page and that is exactly what it returns.

Let us walk through the first three rungs of that ladder by hand, in painful detail, so you can see exactly what the calculator does.

Rung 1 — computing B(1) from B(0)

We know B(0) = 1. We want B(1). The recursion says:

```
B(1) = ( A × B(0) ) / ( 1 + A × B(0) )
```

Plug in the numbers. A is 5. B(0) is 1.

```
Top: A × B(0) = 5 × 1 = 5
Bottom: 1 + A × B(0) = 1 + 5 = 6
B(1) = 5 / 6 ≈ 0.8333
```

Intuitive check: with only 1 phone line and 5 Erlangs of load coming at it, of course it is busy almost all the time. 83% blocking sounds about right.

Rung 2 — computing B(2) from B(1)

Same recipe, but now k = 2 and we use B(1) ≈ 0.8333.

```
Top: A × B(1) = 5 × 0.8333 ≈ 4.1667
Bottom: 2 + A × B(1) = 2 + 4.1667 ≈ 6.1667
B(2) = 4.1667 / 6.1667 ≈ 0.6757
```

Two lines, 67% blocking. Still bad, but better than one line.

Rung 3 — computing B(3) from B(2)

```
Top: 5 × 0.6757 ≈ 3.378

Bottom: 3 + 3.378 ≈ 6.378

B(3) = 3.378 / 6.378 ≈ 0.5297
```

Notice the pattern. Each new line drops the blocking probability — fast at first, then slower. By the time we reach B(8) ≈ 0.07, we have a comfortable margin. Each additional line is buying us less and less improvement, which is a real-world signal: at some point, adding more trunks stops being worth the money.

That is the entire Erlang B calculation. Eight rungs of three-line arithmetic. The Tools page does this in microseconds, but it is arithmetic you could do with pen and paper in five minutes.

> *A useful exercise: pick any A and N, do the recursion on paper, and check against the Tools page. They will match every time.*

5 · Erlang C — When Callers Queue Instead of Hanging Up

5.1 What Changes

Contact-centre callers do not hang up the moment all agents are busy. They wait. They listen to your hold music. So you need a different model: one that asks not "what is the chance of being blocked?" but "what is the chance of having to wait at all, and if I wait, how long?"

That is Erlang C. It is built directly on top of Erlang B.

Think about the difference between a public phone box and your bank call centre. The phone box: if it is in use when you arrive, you walk away. The bank call centre: if all the agents are busy when you call, you sit on hold listening to lift music. Same load on the system, completely different mathematics, because the second model has queueing built in.

Why does queueing change the maths? Because in the phone-box model, all the load that arrives during a busy moment vanishes — those callers just leave. In the call-centre model, none of it vanishes — every call sits there waiting, and the queue builds up until enough agents free up to clear it. The queue itself becomes a thing to think about: how long is it, how long do people wait in it, how often does somebody actually have to join it.

5.2 The Probability of Waiting

The starting question: what is the probability that a brand-new call finds every agent busy and therefore has to wait? Call that C.

```
C(N, A) = N × B(N, A)
```

```
N – A × ( 1 – B(N, A) )
```

It is built directly from B(N, A) — exactly the recursion you just saw. That is why every Erlang C calculator quietly computes Erlang B first.

Let us pick this formula apart piece by piece.

The top of the fraction is N × B(N, A). N is the number of agents. B(N, A) is the blocking probability we worked out in Section 4. So the top is "how many agents you have, times the chance the system is full." This represents how the Erlang B answer plugs in.

The bottom is more interesting. It has two pieces:

- N — the number of agents you have.
- A × (1 – B(N, A)) — the load that is actually being served (load times the fraction of calls that get through). Because A is the offered load and B is the share of calls turned away, (1 – B) is the share that go through, and A × (1 – B) is the effective load running on your agents.

So the bottom of the fraction is "agents minus effective load." That is the spare capacity — the amount of headroom you have. Bigger headroom means smaller chance of waiting. The whole formula is saying: "the chance of waiting is proportional to how full the system gets, divided by how much headroom remains."

You will hit a problem with this formula if N ≤ A. In that case the bottom of the fraction goes to zero or negative, and the formula blows up. That is the mathematics screaming at you: with fewer agents than the average load, the queue grows forever and no steady-state answer exists. The Tools page checks for this and returns 100% queueing in that case — which is operationally correct: every caller will end up waiting if you are understaffed at the average level.

5.3 What Happens After You Wait

Once you have C, the rest of the questions answer themselves. They all use one more idea: the spare capacity. If you have N agents and A Erlangs of load, your spare capacity is (N – A). The bigger that gap, the faster the queue drains.

Spare capacity is worth dwelling on, because it does the heavy lifting in every single formula that follows. Picture a bathtub. Water is coming in from the tap at A litres per minute. Water is leaking out of the plughole at N litres per minute. As long as N is bigger than A, the tub will never overflow. But how fast does the water level fall when somebody briefly puts a bucket of water in? That depends on the gap (N – A), the drain rate minus the fill rate. The bigger the gap, the faster the water level recovers.

Our queue works the same way. Calls flow in at average rate corresponding to A Erlangs. Calls flow out as agents finish them. When a clump arrives and creates a temporary queue, the rate at which that queue drains depends on (N – A). Tight staffing means a small gap and a slow drain. Generous staffing means a big gap and a fast drain. That is why every queue formula has (N – A) in it.

Service level — the chance of being answered within time T

```
SLA(T) = 1 – C × e ^ ( – (N – A) × T / AHT )
```

In words: of the people who had to wait, what share have been picked up by time T? The "e to the power of" piece is a decay curve: the longer T, the more of the queue has cleared.

Let us decode that formula one piece at a time.

Start with C. That is the probability of having to wait at all (from the previous formula). If C = 0.20, then 20% of callers find every agent busy and have to queue. The other 80% are picked up instantly.

Now look at the e^(–...) piece. e is the special number from the vocabulary section (about 2.72). When you raise e to a negative power, the result is a number between 0 and 1. The bigger the negative power, the closer to 0. Concretely: e^0 = 1, e^(–1) ≈ 0.37, e^(–2) ≈ 0.135, e^(–3) ≈ 0.05. The exponent – (N – A) × T / AHT gets bigger (more negative) when T is larger, when spare capacity is larger, or when AHT is smaller. So this whole piece tells you: out of the unlucky people who had to wait, what fraction are still waiting at time T.

Multiplying C by that decay factor gives: "the share of all callers who are still waiting at time T." Subtracting from 1 flips that into: "the share of all callers who have been answered by time T." That is your SLA.

Worked example: suppose C = 0.20, N – A = 3, T = 20 seconds, AHT = 180 seconds. The exponent is –3 × 20 / 180 = –60/180 = –0.333. So e^(–0.333) ≈ 0.717. Multiply: 0.20 × 0.717 = 0.143. Subtract from 1: SLA = 1 – 0.143 = 0.857. So 85.7% of callers will have been answered within 20 seconds. That is your service level.

Average speed of answer — across everybody

```
ASA = C × AHT / (N – A)
```

Most callers get picked up straight away (probability 1 – C). The ones who do wait, wait an average of AHT / (N – A) seconds. Multiply by the fraction who actually wait and you get the average across everybody. That is your ASA.

To see why AHT / (N – A) is the average wait for the unlucky ones, think back to the bathtub. Each call ahead of you in the queue takes AHT seconds to finish on average. There are (N – A) agents who are effectively "draining" the queue at any moment. So your wait depends on how long it takes the drainers to clear whatever is ahead of you. AHT / (N – A) is the average answer that falls out of this picture.

Worked example: keep C = 0.20, AHT = 180, N – A = 3. Wait for the unlucky = 180 / 3 = 60 seconds. ASA = 0.20 × 60 = 12 seconds. Most callers wait nothing; the unlucky 20% wait an average of a minute; overall, across everybody, the average wait is 12 seconds.

Queue time for the unlucky ones

```
Average queue time (if you queue) = AHT / (N – A)
```

This is the average wait, in seconds, for the people who actually had to wait. Notice it does not depend on C — once you are in the queue, how long you wait depends only on the drain rate, not on how often people end up queueing in the first place. ASA blends this with the people who got through instantly; this number is the experience of just the unlucky ones.

Average queue size

```
Queue size = C × A / (N − A)
```

How many callers are sitting on hold at a typical moment. With C = 0.20, A = 5 and N – A = 3, the average queue size is 0.20 × 5 / 3 ≈ 0.33 callers. That sounds tiny, but remember this is averaged across all the moments — most moments have nobody waiting, and a few moments have several waiting. The average comes out small.

Abandonment

```
Abandon% after time t = C × e ^ ( − (N − A) × t / AHT )
```

Exactly the SLA formula in reverse — the share of the queue that has not been answered by time t and would, in this simple model, give up.

Compare it to the SLA formula. SLA is "1 minus this thing." So if SLA at 20 seconds is 85.7%, then the share still waiting at 20 seconds is 100% – 85.7% = 14.3%. If your model says a caller will abandon if they have been waiting longer than 20 seconds, then 14.3% is your predicted abandon rate. The maths is the same as for the SLA; we just stop short of subtracting from 1.

Important: this is a very crude abandonment model. Real callers do not all hang up at exactly the same wait threshold. Some are patient, some are not. The formula here gives you a rough estimate assuming everybody has the same patience. Treat it as a sanity check, not a gospel forecast.

Occupancy

```
Occupancy = A / N
```

How busy your average agent is. With 5 Erlangs of work and 8 agents, each agent is busy 5/8 of the time, or 62.5%. Too low and you are paying for idle staff. Too high and your team burns out. Most contact-centre operations target 75–85%.

The reason this is just A divided by N is straightforward. A is the total number of "agent-hours of work" that arrived during the hour. N agents were available for one hour each. So the fraction of agent time that was spent actually working is A divided by N. If A = 5 and N = 8, then 5 of the 8 agent-hours got used and 3 went to idle time. 5/8 = 62.5%, which is the occupancy.

Occupancy is the metric that creates the biggest tension in a contact-centre operation. Finance wants it high (paying for idle agents is expensive). Operations wants it modest (too high a number means agents have no recovery time between calls, quality drops, attrition climbs). The Erlang model lets you read it directly off the staffing decision and have the conversation with both sides using the same numbers.

6 · How the "Agents Required" Calculator Actually Decides

This is the most-used calculator on the page, and it works by trying. There is no closed-form formula for "how many agents do I need." There cannot be — N is a whole number, and the relationship between N and SLA is bumpy.

So the calculator does something simple. It says: start with the smallest plausible number of agents (one more than the raw load A, because anything less than A can never keep up). Compute the SLA for that many agents. If it meets the target, stop. If not, add one agent and try again.

A few words on what "ceil" means. It is short for "ceiling" — round up to the next whole number. So ceil(5.0) = 5, ceil(5.1) = 6, ceil(7.9) = 8. We use it on the load A because A is usually a decimal but the number of agents has to be a whole number.

Why start at ceil(A) + 1 and not ceil(A)? Because, as we said earlier, staffing at exactly the average load is not survivable — the system has no spare capacity to absorb clumps and the queue grows forever. So the smallest plausible candidate is one more than ceil(A). For A = 5, the calculator starts trying at N = 6. For A = 12.4, it starts at N = 14.

The "while loop" then keeps adding one agent at a time, recomputing the SLA each time, until the SLA meets or beats the target. Each pass through the loop is one more application of: run the Erlang B recursion, derive C, plug into the SLA formula, compare to target. Three lines of arithmetic, repeated maybe ten or twenty times. The whole thing finishes in less than a millisecond.

```
N = ceil(A) + 1
while SLA(N, T, calls, AHT) < target:
N = N + 1
return N
```

That is the whole algorithm. Every time it tests a value of N, it runs the Erlang B recursion to get B(N, A), then computes C(N, A), then the SLA formula. Three lines of arithmetic, repeated until the answer is good enough.

The "fractional agents" output uses the same loop but does one more step: it linearly interpolates between the SLA at N−1 (just below target) and the SLA at N (just above target) to estimate what fraction of an agent you really needed. This has no operational meaning — you cannot hire half a person — but it is useful when costing engagement sizes.

7 · Extended Erlang B — Counting the People Who Try Again

Standard Erlang B assumes that a blocked caller hangs up and never calls back. That is a lie. In real life, somewhere between 10% and 50% of blocked callers redial immediately. When they redial, they add to the load. The extra load means more blocking. More blocking means more redials. The system chases its own tail.

Picture a kettle boiling water. Steam goes up; some condenses on the lid and drips back into the kettle; that water then has to be boiled again, producing more steam, some of which drips back again. The water level will settle at some point, but only after several rounds of evaporation-condensation. Extended Erlang B does the same thing with phone calls. Some calls get blocked, some of those callers redial, those redials add to the load, that creates more blocking, and the cycle goes on until the system

reaches a steady state.

The Extended Erlang B handles this with a small fixed-point loop:

```
start with the original load A
repeat:
compute B = erlangB(N, A)
new_A = original_A + original_A × B × retry_share
if new_A is close to A, stop
A = new_A
return B
```

Each pass adds the retried-calls portion back into the offered traffic, recomputes blocking, recomputes how much extra retry traffic that creates, and stops when the system settles. It usually settles in five or six passes. The result is always higher blocking than plain Erlang B — retries do nothing but make the busy line busier.

A few words on "fixed point" — this is a piece of vocabulary worth owning. A fixed point is a value where the input equals the output. If you keep applying the same rule and you eventually reach a value that the rule does not change anymore, you have found a fixed point. Our loop keeps adjusting the offered traffic A; when an extra pass no longer changes A, we have arrived at the fixed point and we stop. That is the entire idea.

Worked example for intuition. Suppose A = 5 Erlangs, N = 8 trunks, and 20% of blocked callers retry. First pass: B(8, 5) ≈ 0.070, so retried traffic = 5 × 0.070 × 0.20 = 0.07 Erlangs. New A = 5.07. Second pass: B(8, 5.07) ≈ 0.073, retried = 5 × 0.073 × 0.20 ≈ 0.073. New A = 5.073. Third pass: barely changes. We have reached the fixed point. The blocking probability comes out a hair higher than the plain Erlang B answer — about 7.3% instead of 7.0%. With higher retry rates and tighter capacity, the gap gets much bigger and the iteration matters more.

8 · Engset — When the Caller Pool Is Small

Erlang B assumes there are infinitely many possible callers. For a public phone network, that is fine. For a small office with twenty extensions trying to grab one of five outside lines, it is wrong.

Why does the size of the caller pool matter? Imagine an office with only three employees and two outside phone lines. If two of the three are already on calls, the only person left who can possibly start a new call is the third employee. The "pressure" on the system is lower than in a model where infinitely many strangers could be placing calls at any moment. Standard Erlang B does not see this — it treats the caller pool as bottomless. Engset does see it.

Engset corrects for this. If you have M callers and one of them is already on a line, only M−1 can place a new call. That changes the maths slightly:

```
B(N, M, A) = C(M-1, N) × A^N
```

```
sum from i=0 to N of C(M-1, i) × A^i
```

Where C(M-1, i) is the number of ways to pick i busy extensions out of M−1 (a binomial coefficient). The calculator builds the terms one at a time so the binomials never have to be evaluated as huge factorials. For most contact centres you will never need this — Erlang B is close enough — but for private dial-out networks it is the right tool.

A binomial coefficient is just a counting number. C(5, 2) means: out of 5 things, how many different ways can I pick 2? The answer is 10. You can list them if you like — pairs out of {A, B, C, D, E} are AB, AC, AD, AE, BC, BD, BE, CD, CE, DE. Ten pairs. That is what C(5, 2) is shorthand for. The general formula is C(n, k) = n! / (k! × (n – k)!), but we never compute it that way in practice because the factorials get huge. The Engset calculator builds each term from the previous one using the same trick the Erlang B recursion uses — multiply by something small, divide by something small, no factorials needed.

Concrete picture for Engset: imagine an office of 10 employees and 3 shared outside lines. Each employee makes calls only occasionally. You want to know how often somebody picks up the receiver and finds all 3 lines in use. The standard Erlang B would over-estimate the blocking, because it pretends an unlimited number of strangers could be calling. Engset gives the more accurate answer for this small, closed population.

9 · Does the Page Actually Simulate Anything?

No. And it is worth being clear about that, because the original spreadsheet add-in did not simulate either.

There are two ways to answer a queuing question:

- Simulate: write a little program that pretends to receive calls one by one over an imaginary hour, with random arrival gaps and random handle times, and watch what happens. Repeat the hour ten thousand times. Average the results. This is called Monte Carlo simulation, and it is what discrete-event simulators like Arena and SimPy do.
- Compute: assume the arrivals are Poisson and the handle times are exponential, derive the answer in closed form, and use the formula. This is what Erlang B and Erlang C do.

The formulas are massively faster and, when their assumptions hold, give exactly the long-run answer the simulation would converge to. The calculator on the Tools page is the formula route. It can answer in microseconds because there is no random process at all — only the mathematics of the steady-state queue.

"Steady state" deserves a definition. It means the long-run, average-condition behaviour of a system that has been running long enough to forget how it started. If you flip a fair coin enough times, the share of heads settles toward 0.5; that 0.5 is the steady-state value. In our queue, the steady-state values are the average waiting time, the average queue length, the average occupancy — what the system looks like once it has been running for a while. The Erlang formulas describe the steady state directly. They do not describe minute-by-minute fluctuations, only the long-run averages.

A bit more on simulation, since it is the cousin to all this. The word "Monte Carlo" comes from the famous casino — the idea is that you let randomness do the work for you. Roll dice (or have a computer roll fake dice) enough times, and the patterns you see will be the real answer. For a queue: imagine generating a fake hour by drawing random numbers for "time until next call" and "duration of next call," then walking through the hour second by second. Do that thousand of times. Average the answers. You get the same numbers Erlang gives you in closed form. The difference is that the simulation can handle weird cases — agents who take coffee breaks, callers with different patience, multi-skill routing — that Erlang cannot.

That speed is also the cost. Real intervals have non-Poisson arrivals, callers who abandon at very different patience levels, agents with different skills, and breaks. For any of those, you need a real simulator. Erlang gives you the right answer for the textbook case, in a hundredth of a second.

10 · A Worked Example, End to End

Take a small but realistic case. A retail bank inbound queue receives 300 calls per hour. Average handle time is four minutes (240 seconds). The target SLA is 80% of calls answered within 20 seconds.

Before we touch any formulas, let us name what we have and what we want. We have: calls per hour = 300, AHT = 240 seconds, SLA target = 0.80, and the time within which that SLA must be met, T = 20 seconds. We want: the smallest whole number of agents N such that the actual SLA at that N is at least 0.80.

Step 1 — Convert to Erlangs

```
A = 300 × 240 / 3600 = 20 Erlangs
```

Multiply 300 calls by 240 seconds each: 72,000 seconds of talking. Divide by 3600 to convert back to hours: 20 hours of talking, per hour of clock time. So the load is 20 Erlangs. That is the average number of conversations happening at any moment during the hour.

Step 2 — Start the agents loop

You cannot survive with fewer than 20 agents, because the load is 20 Erlangs. Start at 21.

Strictly, the calculator starts at ceil(A) + 1 = 21. We round up the load to the next whole number, then add one more agent of headroom, and that is our first candidate. We could try N = 20, but the SLA formula would blow up because (N – A) = 0 leaves no spare capacity. So 21 is the minimum sensible starting point.

Step 3 — Test 21 agents

Run the Erlang B recursion up to k = 21 with A = 20. You get B(21, 20) ≈ 0.165. Then:

```
C = 21 × 0.165 / (21 – 20 × (1 – 0.165)) ≈ 0.793
```

```
SLA(20s) = 1 – 0.793 × e^(–(21–20) × 20/240) ≈ 0.27
```

27% — nowhere near the 80% target. Add an agent.

Let us decode what just happened. With 21 agents and a 20-Erlang load, the system has spare capacity of (N – A) = 1. That is razor thin. The exponent in the SLA formula is –1 × 20 / 240 = –0.0833. e^(–0.0833) ≈ 0.92. Multiply by C = 0.793: 0.793 × 0.92 ≈ 0.73. Subtract from 1: SLA ≈ 0.27, or 27%. The thin spare capacity is doing almost nothing for us — the queue is essentially always present.

Step 4 — Climb until SLA passes the target

Each new agent improves the spare capacity (N – A) and pushes the SLA up sharply. The calculator increments N one at a time. With this particular case, the loop stops at N = 26, where the SLA comes out at just over 80%.

Watch how fast the SLA improves as the spare capacity grows. At N = 21, (N – A) = 1 and SLA ≈ 27%. At N = 22, (N – A) = 2, the exponent doubles, the decay is much stronger and SLA jumps to about 47%. At N = 23, (N – A) = 3, SLA reaches roughly 63%. At N = 24, around 73%. At N = 25, around 78%. At N = 26, the SLA finally clears 80% and the loop stops. Six rungs of the ladder. Each one took a few microseconds on a computer. The whole calculation finishes before you finish clicking.

Notice that you doubled the spare capacity but did not double the number of agents. With 26 agents you have 30% more headroom than the load demands, and the SLA jumps from "almost nobody answered quickly" to "four out of five answered quickly." That non-linear reward is the heart of why staffing decisions are sensitive: a small amount of extra capacity changes the service experience enormously.

Step 5 — Read off the rest

Once N is fixed, every other output falls out of the formulas in Section 5:

- ASA — usually a handful of seconds at this staffing.
- Queue time for the unlucky ones — AHT / (N – A) = 240 / 6 = 40 seconds.
- Average queue size — C × A / (N – A) — a small number, well under one waiting caller at any instant.
- Occupancy — A / N = 20 / 26 ≈ 77%, comfortably inside the practitioner-preferred 75–85% range.

Let us walk through one of those reads in full. The "unlucky" callers — the ones who actually had to wait — wait an average of AHT / (N – A) seconds. AHT is 240 seconds. N – A is 6. So 240 / 6 = 40 seconds. That is the average wait for the people who did not get an instant answer. Notice it has nothing to do with how often queueing happens; it only depends on the drain rate of the queue. If your operations head says "the people who waited, waited about forty seconds on average," that is consistent with this maths.

And the occupancy. A = 20, N = 26, so each agent is busy 20/26 = 76.9% of the time on average. Translation: out of every hour each agent spends on the floor, they are on calls for about 46 minutes and idle (or doing paperwork between calls) for the other 14 minutes. That is a sustainable working rhythm. Push N down to 22 and occupancy climbs to 91% — agents are barely off one call before the next one lands, the workload feels relentless, quality drops, attrition climbs. The Erlang model lets you see this trade-off as a number, not a feeling.

The calculator on the Tools page does every step above for you in a few microseconds. Every number on the screen has come out of the simple arithmetic in this document — nothing more.

11 · What These Formulas Will Not Tell You

Erlang is a hundred years old and it is a very tidy world. Real contact centres are messy. Things the calculator does not handle:

- Multi-skill agents who can take calls from several queues with different priorities. Erlang assumes everyone is the same.
- Schedule shrinkage — breaks, training, attrition, sick days. Erlang gives you the agents needed on the phone; you still have to inflate that to a roster headcount by dividing by (1 − shrinkage).
- Real abandonment behaviour. Different callers have different patience. The single-number abandon-time approximation here is a rough estimate.
- Non-stationary intervals. If the call rate changes inside the interval you are modelling, Erlang gives the average and misses the spikes. Use shorter intervals to compensate.
- Outbound, blended, or back-office work. None of these are queueing-theoretic in the Erlang sense.

For all of these you need a real simulator. Erlang is the right tool for the textbook inbound queue. It is the wrong tool for everything else — but it is the wrong tool that gets you 90% of the right answer in 0.1% of the time, which is why it has been the contact-centre staple for a hundred years.

One more piece of common sense before the cheat sheet. The Erlang output is a starting point, not the final headcount. If the model says you need 26 agents on the phones, your actual roster has to be bigger — because at any moment some agents are on breaks, some are in training, some are on leave, some are off sick. If 25% of paid agent time gets eaten by these non-productive hours (a typical figure), then a 26-agent requirement translates to roughly 26 / (1 − 0.25) ≈ 35 paid agents on the roster. This inflation step is called "shrinkage adjustment" in WFM jargon and it is the bridge between the Erlang calculator output and a real hiring decision. The Tools page gives you the agents-needed-on-phone number. You have to do the shrinkage step yourself.

In short: Erlang tells you the minimum agents needed at the phones to meet service in one specific interval. To turn that into a real roster, you need to (1) run it for every interval of the day, (2) pick the highest-staffed interval as the headcount peak, (3) inflate by your shrinkage factor, and (4) figure out how to schedule shifts so that the people are on the phones during the right intervals. That last step is the hardest one, and it is where real workforce management actually lives.

12 · One-Page Cheat Sheet

The load

```
A = Calls per hour × AHT seconds / 3600
```

The recursion (compute once, use everywhere)

```
B(0, A) = 1

B(k, A) = A × B(k−1) / ( k + A × B(k−1) )
```

Probability of waiting

```
C(N, A) = N × B(N) / ( N − A × (1 − B(N)) )
```

Service level

```
SLA(T) = 1 − C × exp( −(N − A) × T / AHT )
```

Average speed of answer

```
ASA = C × AHT / (N − A)
```

Queue time for callers who wait

```
Wait = AHT / (N − A)
```

Average queue size

```
L = C × A / (N − A)
```

Abandonment after time t

```
Abandon = C × exp( −(N − A) × t / AHT )
```

Occupancy

```
Occ = A / N
```

Agents required (loop)

```
N = ceil(A) + 1; while SLA(N) < target: N += 1
```

Every number on the Tools page is the output of one of these nine lines. There is nothing else.